Diseño e implementación de un sistema de ficheros sobre una base de datos NoSQL

Alvaro Agea Herradón

Tutor: Francisco Javier García Blas

Co-Director: Daniel Higuero Alonso-Mardones

À vaincre sans péril,
on triomphe sans gloire

- Pierre Corneille

Agradecimientos

En primer lugar, quería agradecer la ayuda a mi tutor Javier, y especialmente a Daniel, sinceramente creo que sin tu ayuda hubiera sido casi imposible haber entregado este proyecto, el esfuerzo por tu parte ha sido enorme y me has llevado en muchos casos a volandas para que pudiera finalizarlo, muchas gracias te lo digo muchas veces pero no me cansaré de repetirlo eres muy grande colega.

También quería agradecerle a mi mujer por todas las noches en vela que la he dado y por aguantarme en esos momentos en donde me subía por las paredes. Espero que sepas llenar el tiempo que voy a tener disponible a partir de ahora.

A mí familia, padres y hermana, por presionarme lo justo para que entregará de una vez el proyecto, pero dejándome en paz cuando yo ya me estaba auto-flagelando lo suficiente. Ya está, al fin.

A mis compañeros de universidad, especialmente a Raúl y Mike, porque me han acompañado en este viaje y hemos disfrutado siempre que hemos podido los éxitos y los fracasos de cada uno.

Por último, a Amazon por el bono de horas que me dio para usar su servicio de manera gratuita para el desarrollo de este proyecto.

Tabla de contenidos

Lista de imágenes

Lista de Tablas

1 Introducción

En este documento se describirá el proceso de creación y diseño del sistema RogerFS, un sistema de ficheros multiplataforma que utiliza una base de datos NoSQL como sistema de almacenamiento.

1.1 Motivaciones del proyecto

Desde principios del siglo XXI con la aparición de los grandes servicios de Internet la gestión de la información se ha convertido en un problema más importante. Es en ese momento, cuando los grandes servicios de búsqueda como Yahoo o Google, empiezan a buscar soluciones para ser capaces de gestionar el volumen de información que ya se encuentra distribuida en Internet y la cual se multiplica cada año. Estas empresas necesitan soluciones que permitan el almacenamiento y el procesamiento de grandes cantidades de información. Por lo que desarrollan tecnologías capaces de distribuir la información en cientos de nodos y con la capacidad de escalar horizontalmente.

Sin embargo, con la aparición de la Web 2.0 y las primeras redes sociales este problema se generaliza, ya no sólo los grandes buscadores tienen problemas para manejar su información, sino que la información que gestionan estas empresas genera esos mismos problemas y necesitan el uso de estas nuevas tecnologías.

Por otra lado, este problema empieza a preocupar a todas las empresas del sector de la TIC (Tecnologías de la Información y la comunicación). El uso de estas tecnologías les ofrece la oportunidad de gestionar información que hasta ahora no estaban tratando debido a los costes que suponía su almacenamiento y procesamiento.

La problemática que supone la gestión de grandes cantidades de datos se empieza a denominar como el problema del Big Data, y las tecnologías que dan soluciones a diferentes aspectos de este problema han sido

englobadas en un nuevo ecosistema tecnológico con tres características principales:

1. **Escalabilidad horizontal.** Al añadir más maquinas en el sistema las capacidades del mismo se multiplican, permitiendo de esta forma que el servicio crezca según va creciendo el servicio.
2. ***Commodity hardware*[1].** Hardware que no ha sido hecho a medida, y que puede ser adquirido en mayoristas.
3. **Tolerante a fallos.** Debido a que el hardware donde se despliega el sistema no es especializado, se considera que la probabilidad de que el uso intensivo del sistema pueda producir fallos es muy alta, por lo tanto el software debe ser capaz de sobreponerse a diferentes tipos de fallos.

En la actualidad, los problemas Big Data ya no son sólo únicamente un problema de volumen, sino que requieren que la información sea almacenada rápidamente desde diferentes fuentes de información. Estas fuentes normalmente son fuentes semi-estructuradas, es decir, información que es estructurada, sin embargo esta estructura no es homogénea y, normalmente, es dependiente de la fuente original. Por último, esta información no está normalizada, por lo que normalmente no se puede almacenar en una base de datos sin realizar un procesamiento previo.

Para resolver este problema, esta información en bruto es almacenada en un sistema de ficheros. Sin embargo, estos sistemas de ficheros no están pensados para que los usuarios finales consuman directamente la información, debido, principalmente, a sus alta latencias.

Para que el usuario pueda consumir la información, se necesita una base de datos que tenga una estructura de datos flexible y que permita el acceso a la información con baja latencia, como por ejemplo las base de datos NoSQL.

---

[1] El *commodity hardware* no implica que sea hardware viejo o desechable,

Esta situación provoca que las arquitecturas actuales utilicen una jerarquía de almacenamiento con dos niveles: nivel online y offline[2].

- **Nivel online u operacional.** Este nivel proporciona un servicio continuo con un acceso a los datos de baja latencia. La información esta estructurada y preparada para ser consumida directamente por el usuario final.
- **Nivel offline o analítico.** Este nivel proporciona un almacenamiento de largo plazo al sistema y no tiene la capacidad de responder a una alta tasa de consultas. Este nivel es utilizado para persistir información sin tratar, también denominada RAW. Esta información no es consumida directamente por el usuario final, sin embargo es utilizada para calcular la información expuesta por el nivel online.

El stack tecnológico actual requiere que los desarrolladores utilicen una tecnología diferente para cada nivel de almacenamiento, esto provoca problemas de mantenimiento y que el proceso de actualización del nivel online sea complejo y laborioso.

1.2 Objetivos del proyecto

El objetivo principal de este proyecto es la creación de un sistema de ficheros distribuido sobre una base de datos NoSQL. De esta forma se podrá reducir el conjunto de tecnologías utilizadas en la arquitectura, simplificando el trabajo entre la diferentes capas de la arquitectura.

Además de este objetivo principal, este proyecto debe satisfacer los siguientes objetivos.

- El sistema de ficheros debe estar diseñado para que pueda ser distribuible y escalable.
- El sistema de ficheros debe poderse adaptar a diferentes bases de datos NoSQL.

[2] Big data explained. https://www.mongodb.com/big-data-explained

- El sistema de ficheros debe ser compatible con algún framework de procesamiento distribuido.
- El sistema debe dar mecanismos para poder copiar información desde el sistema local de ficheros.

1.3 Estructura del documento

En este documento podemos encontrar todos los detalles de diseño e implementación del sistema RogerFS. La estructura de este documento pasa por las diferentes fases desde su concepción inicial, hasta la pruebas finales del sistema:

- **Estado del arte.** En esta sección se analizará la problemática general y de las tecnologías implicadas en este proyecto.
- **Requisitos del proyecto.** En este se capítulo se describirán los requerimientos de diseño y funcionales del proyecto.
- **Diseño.** En este apartado se detallará la implementación del proyecto y las decisiones tomadas durante el proceso.
- **Evaluaciones y pruebas.** Para poder evaluar la adecuación de los requisitos se han realizado un serie de pruebas para validar los resultados de los proyecto.
- **Conclusiones.** En este apartado se describirán las conclusiones extraídas durante el desarrollo del proyecto. Además se detallarán los trabajos futuros que dan continuidad a este proyecto.

2 Estado del arte

Desde la publicación de Map Reduce en 2004 (Dean & Ghemawat, 2004), Google sentó las bases de lo que sería un nuevo ecosistema tecnológico. Éste respondía a la necesidad del mercado de encontrar una solución eficaz al problema del tratamiento masivo de datos, o como después se conocería como el problema del Big Data.

A partir de ese momento empezaron a aparecer tecnologías relacionadas, que intentaba resolver este problema y sus posibles variantes, en lo que comenzó a conocerse como ecosistema Big Data. Poco después, del artículo de Google, apareció un proyecto que vendría a revolucionar la industria del procesamiento de datos Apache Hadoop[3], este proyecto consistía, en aquel momento, de dos sub-proyectos: Apache Hadoop MapReduce, una implementación de la publicación de Google (Dean & Ghemawat, 2004), y Hadoop Distributed File System (HDFS), una implementación basada en el sistema descrito por la publicación Google File System (Sanjay , Howard , & Shun-Tak , 2003). Posteriormente se añadió otro sub-proyecto Apache Hadoop YARN, un sistema de planificación y de gestión de recursos de cluster.

Aunque éste es el proyecto más famoso, pronto la comunidad empezó a crear otros proyectos relacionados como Apache Cassandra[4], Apache HBase[5] o Apache Spark[6], en este primer apartado hablaremos en mas detalle de todas estas tecnologías y definiremos la base tecnológica utilizada para este proyecto.

[3]Web del proyecto Apache Hadoop (https://hadoop.apache.org/)

[4] Web del proyecto Apache Cassandra (http://cassandra.apache.org/)

[5] Web del proyecto Apache HBase (http://hbase.apache.org/)

[6]Web del proyecto Apache Spark (http://spark.apache.org/)

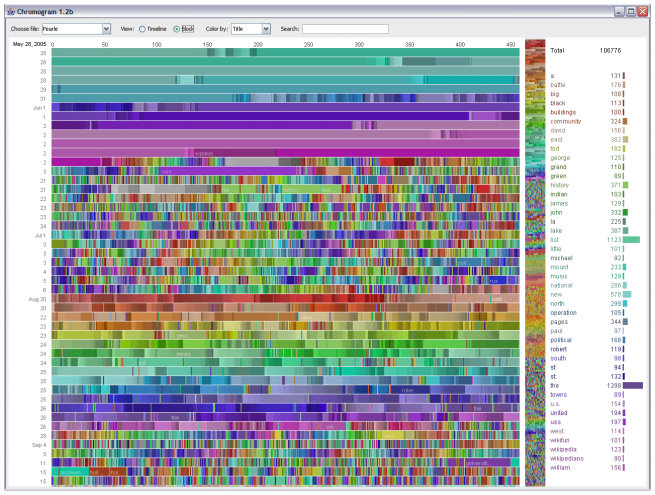

Figure 1. Visualización de la ediciones diarias que se hacen la wikipedia por IBM.

En esta sección vamos a dividir estos proyectos en dos: sistemas de almacenamiento distribuido o base de datos distribuidas como HDFS, HBase o Cassandra, y frameworks de computación distribuida como Hadoop MapReduce o Spark.

2.1 Sistemas de Almacenamiento

Los sistemas de almacenamiento distribuido están orientados a proveer grandes cantidades de información en un almacenamiento escalable, replicado y tolerante a fallos. Los sistemas de ficheros como HDFS no gestionan problemas como la estructura de la información o los metadatos del contenido, sin embargo, las base de datos distribuidas proponen entornos escalables donde el método de acceso y el contenido de la información es importante, y por ello, es gestionado por el motor de la base de datos.

2.1.1 Hadoop Distributed File System

Apache Hadoop es la marca más reconocida del grupo de proyectos englobados dentro del ecosistema de tecnologías Big Data. Es el proyecto mas longevo debido a dos motivos: se liberó en septiembre del 2007 por

parte de Yahoo, y ha sido el proyecto más influyente dentro de la comunidad open source en los últimos años.

Hadoop es una arquitectura de almacenamiento y procesamiento masivo de información, altamente escalable y distribuible. Que se divide principalmente en tres partes:

- **Apache Hadoop MapReduce,** un framework de computación distribuida que permite escalar a cientos de máquina, utilizando el patrón heredado del lenguaje funcional Map/Reduce para procesar de manera distribuida enormes datasets.
- **Apache HDFS,** es un sistema de ficheros distribuidos altamente escalable, capaz de almacenar ficheros del orden de petabytes. Escrito en java utiliza un sistema master esclavo para poder distribuir los ficheros. La interfaz definida por el sistema se ha convertido en un estándar de facto para otros sistemas de almacenamiento distribuido. Otros sistemas como GlusterFS o Tachyon implementan esta interfaz facilitando la integración de otros sistemas a las tecnologías del ecosistema Hadoop.
- **Apache YARN,** es un gestor de recursos de cluster. Cuando un servicio o tarea requiere usar ciertos recursos del cluster, este solicita al master de YARN donde puede alocar los procesos, el se encarga de distribuir la tareas teniendo en cuenta la memoria y el número de cores requeridos para la operación.

Estos tres proyecto dan soporte a las necesidades básica de un proyecto Big Data orientado al procesamiento de grandes cantidades de información, y son la base de la distribuciones más conocidas de Hadoop como CDH de Cloudera[7] o HDP de Hortonworks[8].

Hadoop Distributed File System (HDFS) es un sistema de ficheros distribuido para sistemas commodity, esta preparado para soportar

[7] Relación de Cloudera con Hadoop (http://goo.gl/V7Rzqb)

[8] Open Enterprise Hadoop (http://goo.gl/ldhAEG)

fichero del orden de terabytes de información, entre sus principales características son: una arquitectura master/esclavo, la alta disponibilidad, la inmutabilidad de la información, la replicación y la tolerancia a fallos. Su arquitectura permite la escritura de grandes ficheros y su procesamiento posterior utilizando frameworks de procesamiento distribuido como Hadoop MapReduce o Spark.

La arquitectura básica de HDFS consiste en dos tipos de servicio:

- El *NameNode*, es el nodo maestro, se encarga de la gestión de los metadatos y contiene la información necesaria para encontrar la información en el cluster, es también el encargado de coordinar las operaciones de escritura.
- El *DataNode* es el nodo esclavo, que contiene la información almacenada en el cluster. Los clientes de HDFS se comunican con el para trasmitir los bytes de escritura y de lectura en cada una de las operaciones pero no tienen capacidad de coordinación.

La división de la información se recomienda hacer en particiones de un tamaño mínimo de 64MB, estas particiones son tan grandes porque el objetivo del proyecto es almacenar archivos de gran tamaño y beneficiar la lectura de esto ficheros para su posterior procesamiento.

Por este motivo, en caso de que se quiera introducir muchos archivos de pequeño tamaño el sistema no estará optimizado permitiendo mucho rendimiento. El sistema no valida si la información cumple con un formato concreto, tampoco tiene en cuenta el tipo de información para hacer las particiones, por ejemplo, si nuestro dataset es un CSV, donde cada línea es un registro, HDFS no garantiza que una partición contenga todas las líneas completas, por lo que puede pasar (y pasará) que el fichero cargado quede divido en dos particiones y el sistema de procesamiento tenga que buscar la forma de juntar estas dos partes.

La tolerancia a fallos es primordial en HDFS[9] en su configuración por defecto, la información esta replicada en tres nodos[10] y se incluyen configuraciones para gestionar coordinación de clusters para la replicación de la información multi-datacenter. Esto permite al sistema de almacenamiento proveer tolerancia a fallos y planes de contingencia frente a catástrofes.

Sin embargo, uno de los grandes problemas que ha tenido HDFS ha sido debido a su estricta arquitectura maestro/esclavo. En la arquitectura planteada por Google, y que inspiro Hadoop, el maestro era un punto único de fallo. Sin embargo, en su artículo (Sanjay , Howard , & Shun-Tak , 2003) no se daba importancia a este problema, debido al hecho de que monitorizar una sola máquina frente a los cientos de nodos que conformaban el cluster, era una mejora sustancial a la situación anterior.

Sin embargo, en la practica no sólo se contaba con un cluster Hadoop con un solo maestro. Por el contrario, se desplegaba diferentes clusters con sus correspondientes maestros, además la potencia de estos sistemas se solía combinar en complejos *pipelines*. Por este motivo, la alta disponibilidad del sistema se convirtió en un requisito , esto se materializo en la creación de un nuevo tipo de nodo en HDFS el *Secondary NameNode*. Este sería capaz de convertirse en maestro en caso de que el *NameNode* principal dejara de funcionar, por lo que el sistema contaba con mecanismo de alta disponibilidad activo/pasivo.

---

[9] La tolerancia a fallos y los planes de contingencia frente a catástrofes son muy importante en los sistema Big Data, porque al tener tanta cantidad de información realizar copias de respaldo de la información es una tarea costosa y muy pesada, esto sistemas permiten estar protegido sin necesidad de tener backups completos del sistema.

[10] La información esta replicada en tres nodos porque originalmente se pensó que una replica estuviera en el mismo RACK para protegerse frente a caídas de máquinas y otra replica estuviera en un RACK diferente para protegerse de caídas de todo el armario o del centro de comunicaciones.

Para la gestión de ficheros, HDFS utiliza el concepto namespace que es similar al concepto de directorios de un sistema de ficheros tradicional, estos namespaces permiten la gestión de archivos dentro del sistema y se pueden utilizar como un índice que permita el filtrado de información al nivel de carpeta y de esta forma los sistemas de búsqueda puedan seleccionar la información necesaria de una forma directa.

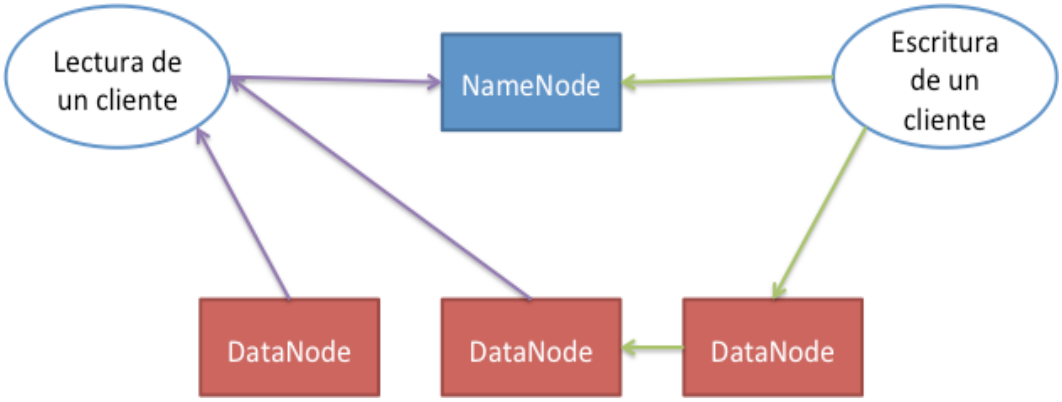

Figure 2. Comunicación en HDFS entre nodos.

La comunicación al ser coordinada por el nodo maestro puede generar un posible cuello de botella, sin embargo, al realizar operaciones ligeras y rápidas, no suele representar un punto de fallo en el sistema. La secuencia de comunicación (véase Figure 2) se realiza en primera instancia contra el maestro, para actualizar los metadatos y gestionar los bloqueos de ficheros[11], y después contra los esclavos o *DataNodes,* en esta segunda comunicación el usuario hace la transferencia de los datos, por lo que la carga pesada la comparten estos nodos.

En conclusión, HDFS es un sistema robusto para el almacenamiento de grandes cantidades de información inmutable y no estructurada, que quiera ser consumida por motores de procesamiento distribuido, sin embargo en caso de que se quiera acceder a la información de una forma

[11] En HDFS un fichero sólo puede ser escrito por un único cliente, en caso de que haya varios cliente la información deberá estar agrupada bajo un mismo namespace.

directa y sobre sectores concretos, este sistema de ficheros no da una capacidad óptima para hacer este procesamiento.

2.1.2 Apache HBase

Apache HBase es una base de datos distribuida orientada a columnas, que en su modo distribuido se apoya en HDFS como sistemas de ficheros distribuidos. Este proyecto está basado en el paper de Google BigTable (Chang, et al., 2006). Que tiene como base tres grandes innovaciones:

1. **El modelo de datos.** Big Table presenta un modelo datos tridimensional, donde la unidad mínima de información es la columna, la cual esta compuesta por tres datos: el nombre de la columna, el dato y el timestamp. Las columnas se agrupan en filas y esta se agrupan en tablas. Cada fila puede tener infinitas columna y dos filas en la misma tabla no tienen porque tener la mismas columnas. Sin embargo, el sistema mantiene el orden de esta columna dentro de la propia fila, lo que permite un acceso rápido a la información de una columna concreta.
2. **El concepto SSTable.** Es un almacenamiento desagregado e inmutable de la información contenida en las tablas, junto con un índice que permite el acceso aleatorio a la información. Esta información guarda los diferentes estados de una columna, y permitiendo almacenar información mutable como inmutable.
3. **Lock distribuido.** BigTable presenta un sistema de lock distribuido que permite la sincronización de ciertas operaciones en el cluster, también permite la creación de contadores distribuidos y la ejecución de operaciones con consistencia.

Apache HBase hereda todas estas características, apoyandose en HDFS para la distribución de sus SSTables a lo largo del cluster.

El objetivo de esta base de datos es la escalabilidad a miles de usuarios en escritura y lectura aleatoria y dar un método para maneja la información de forma inmutable. Por otro lado su fácil integración con motores de

procesamiento como Hadoop hacen que esta base de datos sea muy atractiva a la hora de monitorizar información en tiempo real.

2.1.3 Apache Cassandra

Apache Cassandra se considera un proyecto dentro del ecosistema Hadoop, pero su arquitectura es muy diferente al resto de tecnologías del ecosistema, por lo que se podría considerar que no pertenece al mismo. Las principales diferencias con el resto de tecnologías del ecosistema son las siguientes:

1. **No utiliza como sistema de almacenamiento HDFS.** A diferencia de otros sistemas como HBase, Cassandra no utiliza HDFS como su sistema de almacenamiento, y utiliza un formato propio de información.
2. **No utiliza una arquitectura maestro/esclavo.** A diferencia de la mayoría de los proyectos del ecosistema, Cassandra no utiliza una arquitectura maestro esclavo, sino que utiliza una arquitectura P2P más moderna y robusta.

Por esto motivos esta tecnología ha sido excluida de las principales distribuciones de Hadoop, como Cloudera, Hortonworks o MapR[12], y han aparecido diferentes alternativas de código abierto con funcionalidad similar a Apache Cassandra, como por ejemplo, Apache Falcon[13] o Apache Accumulo[14].

Inicialmente, el proyecto creado por Facebook, quería integrar en una sola tecnología el modelo de distribución descentralizado propuesto por

---

[12] Actualmente la principal empresa encargada del desarrollo, mantenimiento y puesta en producción es DataStax (anteriormente Riptano), su CTO Jonathan Ellis es el Project Chair del proyecto Apache Cassandra (Cassandra, 2015)

[13] Web del proyecto Apache Falcon (http://falcon.apache.org/)

[14] Web del proyecto Apache Accumulo (http://accumulo.apache.org/)

Amazon DynamoDB (DeCandia, et al., 2007) y el modelo de datos propuesto en Google BigTable (Chang, et al., 2006). Mezclando estos sistemas se conseguiría una base de datos tolerante a fallos, altamente distribuible, con escalabilidad lineal y P2P, con un modelo de datos robusto que permite la escritura y lectura simultanea de millones de usuarios a la vez.

Aunque la evolución del proyecto ha hecho que ciertas ideas iniciales cambiaran, como por ejemplo, el modelo de datos se ha alejado de los conceptos iniciales de BigTable y el lenguaje de consulta se ha intentado asemejar al lenguaje de consulta SQL, la parte de distribución de los datos se mantiene prácticamente igual a la planteada por DynamoDB.

DynamoDB propone una base de datos clave/valor linealmente escalable y P2P, esto se consigue haciendo que la información se distribuya en un anillo donde cada uno de los nodos se encarga de una parte de la información. Los datos son almacenados utilizando una clave, con la cual se crea un hash, los nodos en ese anillo virtual se han repartido todas las posibles claves generadas por el algoritmo hash, con lo que la información se reparte por los nodos siguiendo ese mapa de particiones (Véase Figure 3). Este método es muy similar al utilizado por Chord (Stoica, Morris, Karger, Kaashoek, & Balakrishnan, 2001) ya que DynamoDB (DeCandia, et al., 2007) esta inspirado en la tecnología descrita en este artículo. Cuando el usuario quiere buscar información en el cluster debe conocer la clave que quiere buscar dentro de la base de datos, pero al conocer la clave también conoce la hash de esa clave, por tanto conoce inequívocamente donde se encuentra el dato que busca y sólo necesita preguntar por dicha información a la máquina implicada. Esto permite aumentar de manera prácticamente lineal el throughput del sistema, debido a que cuantas más máquinas se añadan en este cada nodo se hará cargo de menos información en el sistema.

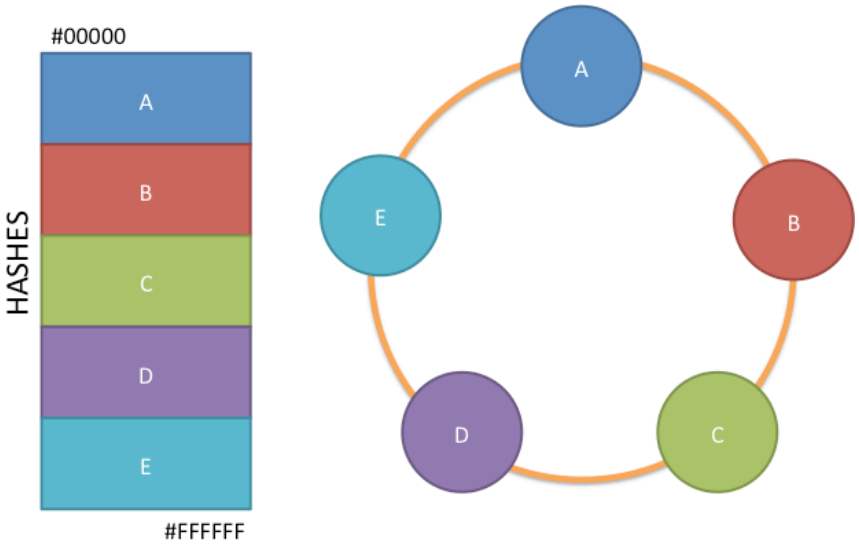

Figure 3. Modelo de distribución de DynamoDB y Cassandra

Aunque el sistema de distribución apenas ha cambiado, el modelo de datos ha sufrido variaciones desde las primeras versiones (Hewitt, 2010). En estas versiones el modelo era muy similar al presentado por BigTable e implementado por Apache HBase, las comunicaciones se hacían a través del protocolo de comunicación Thrift (Thrift, 2015) y el lenguaje era propio y se parecía al propuesto por Google. Sin embargo, con la aparición del Lenguaje CQL3 (Cassandra CQL, 2015) todo esto cambió, se presentó un esquema más parecido al de las bases de datos tradicionales, donde el modelo orientado a columnas se oculta al usuario, aunque sigue siendo la base de la estructura interna de almacenamiento, el lenguaje es más parecido al lenguaje SQL y la comunicación se hace utilizando un protocolo propio dejando el uso de Apache Thrift sólo para la comunicación entre nodos y para sistemas legacy. Otra mejora fue la aparición del soporte nativo para índices secundarios, esto permite al sistema realizar consultas sobre columnas que no fueran la clave de la fila.

Por último, otra característica que diferencia al sistema de otras base de datos NoSQL es su capacidad de ajustar su nivel de consistencia, tanto en lectura como escritura. Cassandra en este caso da múltiples opciones de las que destacan las siguientes:

- **ONE.** En este caso el sistema sólo necesita la confirmación de un nodo antes de dar como buena la escritura o la lectura del usuario. La replicación se realiza posteriormente, pudiendo no producirse en caso de fallo, o porque las replicas no estuviera disponibles.
- **QUORUM.** En esta caso la escritura o lectura necesita que la confirmación (N/2 +1) nodos, donde N es el factor de replicación utilizado en el sistema.
- **ALL.** Este caso es el más estricto y requiere que todos lo nodos donde la información va a ser almacenada confirmen la operación de escritura o lectura.

Para poder garantizar la consistencia de la información almacenada en la base de datos se necesita que el número de nodos que confirmen las operaciones de escritura más el número de nodos que confirmen las operaciones de lectura sea mayor al factor de replicación mas uno. En cualquier otro caso, la información puede ser inconsistente. Esta característica permite optimizar la base de datos para sistema donde se quiera prioriza la escritura frente la lectura o al revés, o optar por sistema más equilibrados donde no es necesario la confirmación de todos los nodos para garantizar la consistencia.

Junto a esto se desarrollaron drivers de procesamiento para frameworks, como Hadoop o Spark para utilizar la base de datos como fuentes de datos. Permitiendo de esta forma poder realizar consultas complejas y a cambio de tener un throughput más reducido.

Estas funcionalidades han hecho de Cassandra una base de datos muy flexible, que es capaz de trabajar en multitud de casos de uso.

2.2 Frameworks de procesamiento

Aunque la información sea almacenada en grandes repositorios de información escalables, el usuario necesita poder acceder a ellos de una forma eficiente e igualmente escalable, para ello se ha desarrollado frameworks de computación distribuida que permiten lanzar procesos de computación en cientos de nodo de manera coordinada y eficiente,

teniendo en cuanta problemas como la localidad de los datos, la tolerancia a fallos o los recursos del cluster.

2.2.1 Hadoop MapReduce

Hadoop MapReduce es un framework que permite escribir aplicaciones que puedan procesar grandes cantidades de datos, del orden de petabytes, en paralelo en cluster de miles de nodos con hardware commodity. Entre las principales características que da este framework es la tolerancia a fallos, la alta disponibilidad y la escalabilidad horizontal de procesos.

Cuando se lanza un trabajo de MapReduce se divide el dataset de entrada en partes independientes, la cuales son procesadas en paralelo por las tareas de Map, estás tareas transforma la fuente original, al igual que la función de Map en una colección. El framework, después de haber aplicado esta transformación, ordena la salida de los Maps y ejecuta las tareas de Reduce. Normalmente tanto la lectura como escritura de los datos se hace desde un sistema de ficheros, aunque también se pueden hacer desde otros sistemas, como por ejemplo una base de datos o un servicio. Hasta la aparición de YARN el propio framework se encargaba de la planificación, monitorización y la repetición de tareas erróneas. Actualmente esto es tarea de YARN aunque la lógica y las tareas de contingencia se encuentran en el propio framework.

Para poder maximizar el rendimiento de las tareas MapReduce, los nodos donde se almacena la información son utilizados como nodos de procesamiento, lo que nos permite aprovechar al máximo la localidad de los datos. Si se usa HDFS (página 20) MapReduce utiliza los mismos nodos para desplegar sus trabajos. Manteniendo la localidad de los datos permite maximizar los recursos, reduciendo la latencia y el ancho de banda, lo que permite que el sistema pueda escalar horizontalmente con facilidad.

El framework MapReduce consiste en un nodo maestro o *ResourceManager*, un esclavo en cada nodo del cluster llamado

NodeManager y un maestro de aplicación o *MRAppMaster* (ver imagen Figure 4).

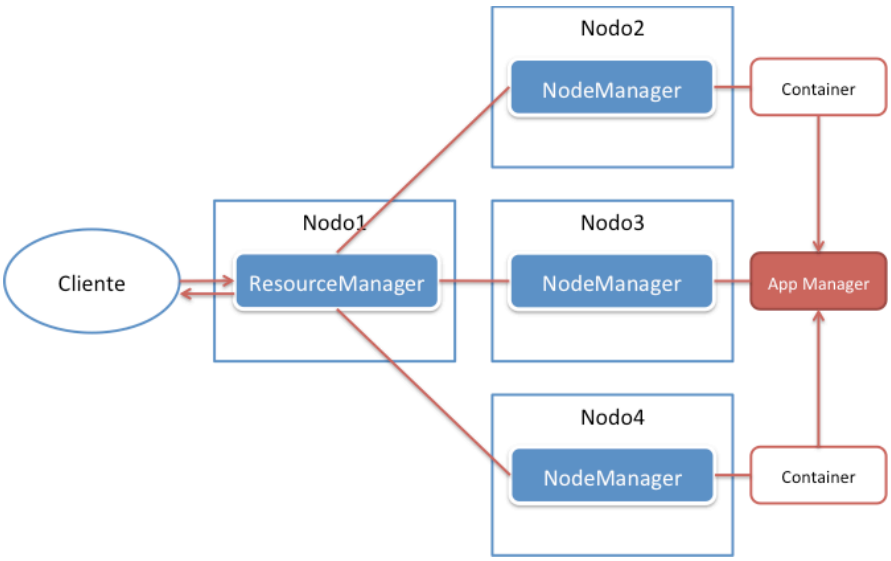

Figure 4. Arquitectura básica de Hadoop Map Reduce

Esta nueva arquitectura permite gestionar más eficientemente los recursos del cluster, además de minimizar las responsabilidades del framework de procesamiento haciéndolo más simple, y por tanto más robusto. Esta arquitectura ha sido tomada como base para las nuevas tecnologías de distribución distribuida.

El flujo de llamadas sería el siguiente, en primer lugar el cliente se pondría en contacto con el *ResourceManager*, pasándole el trabajo a realizar y la configuración de este, el *ResourceManager* analiza este trabajo y comprueba el estado del cluster para poder asignar a la tarea los recursos necesarios (número de cores y memoria), intentando respetar la localidad de los datos. Una vez que los recursos están disponibles el *ResourceManager* designa a una *AppManager,* el cual se encargará de monitorizar a los nodos implicados, y los *Containers* que se encargarán de realizar las tareas. En caso de que haya más clientes en el cluster el *ResourceManager* se encargará de optimizar estos recursos.

Aunque con el cambio de versiones la gestión y monitorización de recursos ha cambiado, en lo esencial Hadoop MapReduce sigue basándose en los mismos enunciados en el artículo de Google.

Estos podrían resumirse en los siguientes:

- El procesamiento de tareas se divide en dos fase principales, la fase de Map, en la cual los elementos son convertidos en objetos clave/valor; y la fase reduce donde se agrupan cada uno de estos objetos por su clave y se agregan (Véase Figure 5)
- Siempre que sea posible se intenta evitar el uso de tráfico de red por lo que se accede a la información de manera local. Para ello el trabajo declara unas localizaciones de preferidas que indican el lugar donde se encuentran los datos en el cluster junto con sus réplicas.
- La información cuando se procesa siempre se escribe en el sistema de almacenamiento, es decir, cada iteración Map/Reduce termina con la escritura de los resultados de salida en el sistema de almacenamiento. De esta forma podemos garantizar el reprocesamiento de los datos en caso de caída de un nodo y utilizamos el propio sistema de ficheros como mecanismo de comunicación entre cada una de las tareas[15].

[15] Esta es una de las características más criticadas del framework Hadoop MapReduce debido a que si almacenamos todas las operaciones intermedias nuestro sistema depende de la latencia del disco, lo cual es una alternativa poco eficiente.

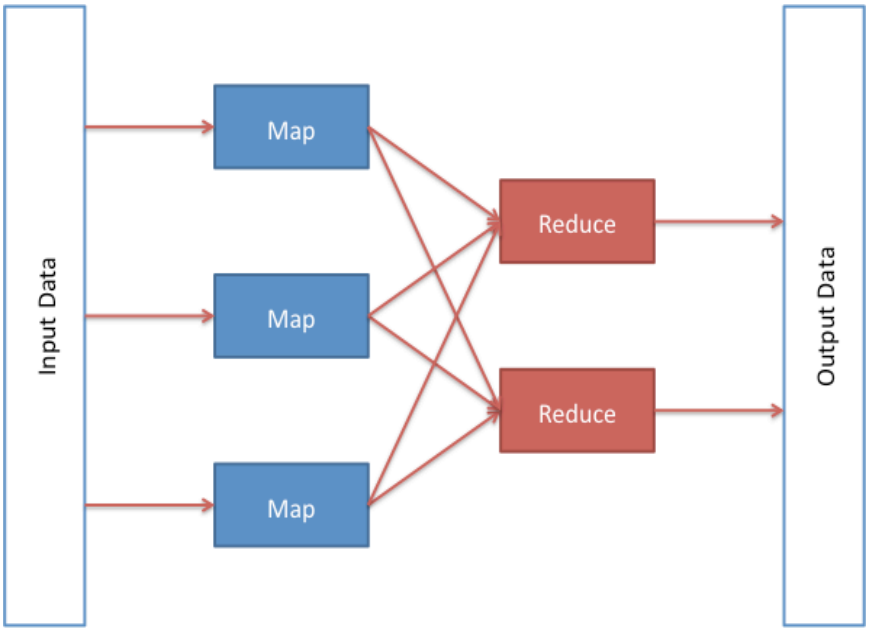

Figure 5. Proceso de MapReduce

Alrededor de este framework de procesamiento a nacido un ecosistema de tecnologías que complementan la funcionalidades de este, como por ejemplo, Hive[16], que permite soporte SQL, Giraph[17], que da soporte para el procesamiento de grafos o Mahout[18], que incluye librerías de machine learning.

2.2.2 Apache Spark

Apache Spark[19] es, tal vez, una de las tecnologías que están más de moda dentro del ecosistema Big Data, debido a que es el líder de lo que se ha llamado la segunda ola del Big Data, en la cuales se incluyen tecnologías como Cloudera Impala[20], Apache Drill[21], Apache Tez[22], Apache Flink[23] y el

[16] Web del proyecto Apache Hive (http://hive.apache.org/)

[17] Web del proyecto Apache Giraph (http://giraph.apache.org/)

[18] Web del proyecto Apache Mahout (http://mahout.apache.org/)

[19] Web del proyecto Apache Spark (http://spark.apache.org/)

[20] Web del proyecto Impala (http://impala.io/)

[21] Web del proyecto Apache Drill (http://drill.apache.org/)

[22] Web del proyecto Apache Tez (http://tez.apache.org/)

[23] Web del proyecto Apache Flink (http://flink.apache.org/)

propio Apache Spark. En la Figure 6 se puede observar como la popularidad de la tecnología ha ido creciendo de manera exponencial en este último tiempo.

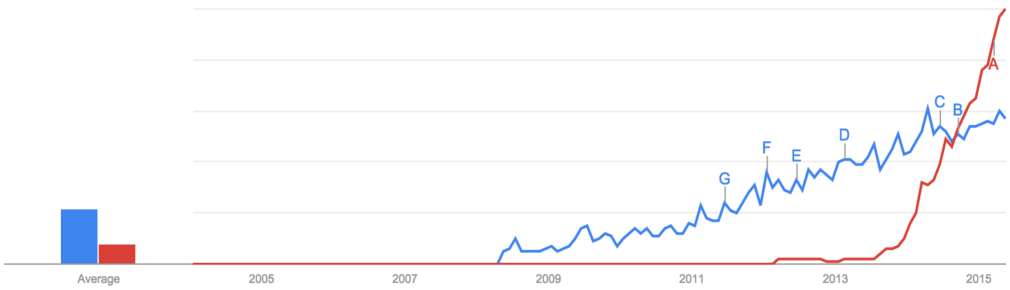

Figure 6. Comparación de búsquedas Apache Hadoop (Azul) y Apache Spark (Rojo)

Esta segunda ola de tecnologías Big Data intenta solucionar problemas propios de la tecnología Hadoop.

- **Utilizar el disco como mecanismo de comunicación entre procesos.** Hadoop utiliza el sistema de ficheros HDFS no sólo como sistema de almacenamiento, también cuando se quieren unir diferentes procesos Map/Reduce se debe pasar por el sistema de fichero para comunicarnos entre cada uno de los procesos. Esto provoca que la latencia del sistema sea muy alta, debido a que el disco lastra el sistema.
- **API demasiado reducida.** El API que propone Hadoop es simple pero no eficaz, si se quieren realizar tareas más complejas el API no da facilidades para trabajar de forma rápida.
- **El concepto MapReduce es funcional, Java No.** La implementación de Hadoop se realizo en Java, en ese momento no contaba con capacidades propias de los lenguajes funcionales[24]. Esto ha provocado que la interfaz de MapReduce este limitada por el propio lenguaje y que la estructuras utilizadas sean más complejas al no adaptarse al paradigma planteado.

[24] La aparición de Java 8 añade características funcionales lo cual a medio plazo mitiga el problema.

- **Mucho código para poca funcionalidad.** Elegir un lenguaje no funcional implica que mucho código se produce para intentar saltar esta limitación que posee el propio lenguaje. Por lo que se produce demasiado código, en su mayoría código repetido.
- **El ecosistema tiene demasiada tecnologías que deben funcionar a la vez.** Debido a que es un sistema multipropósito, el framework Hadoop Map Reduce se puede utilizar para muchos casos, sin embargo, el diseño busca que estos frameworks sean gestionados como servicios independientes, lo cual provoca que se deba gestionar un stack tecnológico muy grande para resolver problemas relativamente sencillos.

Por estos motivos, la comunidad responde y empiezan a surgir diferentes frameworks de procesamiento. Uno de ellos es Spark, el cual es creado dentro del grupo de investigación AmpLab en la Universidad de Berkeley, por Matei Zahaira, el cual sienta las bases del proyecto en su artículo (Zaharia, et al., 2012).

En él, se propone utilizar el disco para las operaciones intermedias y para la comunicación de procesos usar la memoria, lo cual hace que el procesamiento complejo o iterativo sea mucho más rápido.

Para poder dar un API potente se propone usar uno similar a la Dryad Linq (Isard, Budiu, Yu, Birrell, & Fetterly, 2007), usando un lenguaje de programación funcional, como Scala (lenguaje para la JVM) que implementa una mezcla entre lenguaje orientado a objetos y lenguaje funcional (Typesafe, 2015). Con este nuevo lenguaje es más sencillo implementar algoritmos distribuidos y el número de líneas es significativamente menor.

Por último, se define un modelo de datos el RDD (Resilent Distributed Dataset). El RDD es una interfaz que simplifica el acceso a los datos en sistema de almacenamiento distribuido, se compone principalmente de tres partes:

- **Un método de acceso a los datos.** El RDD debe definir un método de acceso a la información al sistema de almacenamiento utilizado, este método además provee mecanismo para optimizar este acceso y reducir el consumo de red, de un forma similar a Hadoop, el almacenamiento además de dar acceso a los datos puede dar información de cómo están distribuidos los datos en el sistema.
- **Un workflow de operaciones.** Este grafo permite saber cuales son las operaciones necesarias para poder procesar la información. Este workflow esta dividido en pasos, estos paso son reprocesables en caso de que el sistema falle se puede reintentar la tarea en otro nodo. El sitema utiliza este workflow para planificar la tareas en el cluster de procesamiento.
- **Un método de escritura de datos.** En caso de que el sistema lo permita, también se podrá realizar escritura en distribuido en el sistema. Este método da acceso a la escritura de información desde los diferente nodos de procesamiento del cluster.

Esta interfaz es común a todo el sistema, por lo que el desarrollo de interfaces particulares se realiza mediante el uso de librerías que extienden las funcionalidades de dicha interfaz. En el caso de utilizar ciertas características concretas, como por ejemplo técnicas de machine learnig o análisis de grafos, no hay que instalar otra tecnología.

En 2012 se presenta el concepto DStream (Zaharia, Das, Li, Shenker, & Stoica, 2012) el cual permite utilizando la abstracción del RDD, procesar información en Streaming y poder mezclar esta información con proceso Batch. En 2013 se anuncia Spark Streaming una nueva API que permite mezclar estos dos tipos de procesamiento utilizando un único framework.

2.3 Resumen

Podemos ver que este stack tecnológico nos permite trabajar en muchos casos de uso debido a la multitud de propiedades con las que cuentan estos sistemas. En las siguientes tablas haremos una comparación a modo resumen de estas tecnologías.

Table 1. Comparativa de sistemas de almacenamiento.

Característica	HDFS	HBase	Cassandra
Basado en…	Google File System	Google BigTable	Dynamo, BigTable
Sistema de ficheros	Si	No	No
Base de datos	No	Si	Si
Acceso aleatorio	Si, a nivel de fichero	Si	Si
Modo de trabajo	Master/Slave	Master/Slave	P2P
Lenguaje de consulta	No	Propio	CQL
Almacenamiento	--	HDFS	Propio
Tecnologías necesarias	Hadoop MapReduce	HDFS, Apache Zookeeper	No
Compatibilidad Hadoop	Si	Si, a través HDFS	Si
Compatibilidad Spark	Si	Si	Si
Tolerancia a fallos	Si	Si	Si
Escalabilidad	Si	Si	Lineal

Por otro lado los sistema de procesamiento distribuido están basados en los mismo conceptos iniciales, sin embargo, al tratarse de tecnologías de diferentes etapas Spark es muy superior actualmente a Apache Hadoop MapReduce.

Table 2. Comparativa de sistemas de procesamiento.

Característica	Hadoop MapReduce	Spark
Basado en...	Map Reduce	RDD
Creación	2007	2012
Tecnologías necesarias	HDFS	Ninguna
Almacenamiento intermedio	HDFS	Memoria
Modo de trabajo	Master/Esclavo	Master/Esclavo
Tipo procesamiento	Batch	Batch/Streaming
Proyectos relacionados	Hive, Giraph, Pig	Oryx, MLlib, GraphX

3 Requisitos del proyecto

En esta sección se hará una disertación de la decisiones de diseño tomadas a la hora de la realización del proyecto y se describirán, en un pequeño resumen, de los requisitos y los resultados esperados del proyecto.

En primer lugar hablaremos sobre los requisitos de arquitectura del proyecto, se plantearán las diferentes alternativas propuestas y cuales han sido la decisiones finales llevadas a cabo en el proyecto. Para ello se presentará la arquitectura con diferentes grados de detalle para ir profundizando cada vez más en el proyecto.

Después se hablará de los requisitos que tendrá la interacción con el usuario para utilizar el proyecto.

Por último, se hará un resumen de los requisitos que detallará el alcance del proyecto.

3.1 Requisitos de Arquitectura

El sistema propuesto en este proyecto será capaz de almacenar gran cantidad de información de forma distribuida y escalable. El proyecto utilizará una base de datos distribuida y extenderá la funcionalidad del sistema para poder gestionar ficheros no estructurados. Por otro lado, esta modificación debe ser compatible con el uso de sistemas de procesamiento distribuido. El sistema propuesto dará al usuario la capacidad de almacenar información no estructurada para su posterior análisis en la propia base de datos NoSQL. De esta forma se facilitará la creación de vistas de la información no estructurada reduciendo el número de tecnologías necesarias.

3.1.1 HDFS + HBase

Esta arquitectura es típicamente planteada en la mayoría de los proyectos Big Data, normalmente el sistema utiliza sistema de ficheros distribuido como un almacenamiento de los datos RAW. La información se guarda sin

procesar, o con un procesamiento ligero, que permita la inserción en el sistema de mucha información con baja latencia. Sin embargo, para permitir acceder a la información a los usuario finales se crean vistas donde esta información está indexada en una base de datos con baja latencia en lectura y con gran capacidad de escalabilidad.

Para almacenar la información RAW se utiliza un sistema de ficheros como HDFS, para poder procesar esta información se necesita un sistema de procesamiento distribuido como Spark o Hadoop, en este caso se elige Spark por su velocidad y rendimiento. Por último, para almacenar las vista se escoge una base de datos NoSQL como HBase.

Este diseño, se puede desplegar con las distribuciones comerciales de Hadoop como Cloudera, Hortonworks o MapR, por lo que es una arquitectura fácil de desplegar.

3.1.2 RogerFS

RogerFS es una abstracción que permite dotar de capacidades de sistemas de ficheros a base de datos NoSQL. Simplificando el desarrollo de aplicaciones que resuelvan el problema de Big Data. Para el desarrollo de este proyecto se han tenido que seleccionar una tecnología de base de datos NoSQL, Apache Cassandra, y un sistema de procesamiento distribuido, Apache Spark.

La opción RogerFS usando como base Cassandra permite minimizar latencias, reducir la complejidad del proyecto, tener una política escalabilidad sencilla y poder aprovechar las características de Cassandra como su despliegue P2P.

En la siguiente tabla (Table 3) se puede ver una comparación entre una arquitectura tradicional usando HDFS y HBase, y la arquitectura propuesta con RogerFS sobre Cassandra:

Table 3. Comparativa entre RogerFS con Cassandra y HDFS + HBase

Característica	RogerFS con Cassandra	HDFS + HBase
Arquitectura	P2P	Master/Esclavo
Tecnologías necesarias	Cassandra	HDFS, HBase, Zookeeper
Monitorización	1	5[25]
Cache en Sistema Ficheros	Si	No
Multiescritura	Si	No
Compatibilidad Indíces	Si	No
Borrado	No	Si

3.1.3 Resumen de requisitos de arquitectura

Una vez presentadas las alternativas, y las consideraciones de diseño tomadas por ambos conjuntos de tecnologías, en este capítulos se va a realizar un pequeño resumen de los requisitos que se deben tener en cuenta en el desarrollo del proyecto.

ARQ-01. Sistema de ficheros basado en una Base de Datos distribuida.

El sistema implementará un sistema de ficheros utilizando como soporte de almacenamiento una base de datos distribuida. Esto permitirá al usuario utilizar los parámetros de configuración de la base de datos como los únicos parámetros necesarios para la configuración del sistema de ficheros.

[25] Los servicios monitorizados son NameNode, DataNode, HBase Server, Region Server y Zookeeper node, se ha ignorado el Secundary NameNode y Zookeeper Slaves.

ARQ-02. Minimizar el número de tecnologías necesarias.

El proyecto deberá reducir el número de tecnologías necesarias para el despliegue del proyecto, de esta forma también se reducirán los servicios que se necesitan monitorizar.

ARQ-03. Ser compatible con sistemas de procesamiento distribuido.

La información almacenada podrá ser procesada con sistemas de distribuidos como Apache Spark. Para ello se deberán proveer métodos que permitan su procesamiento y faciliten su acceso.

ARQ-04. Almacenar cualquier tipo de fichero.

El sistema deberá ser capaz de almacenar cualquier tipo de fichero, sin importar el formato. Para ello almacenará la información en RAW, sin tener en cuenta codificaciones o sistemas de compresión.

ARQ-05. Permitir compresión en los ficheros.

Se podrá almacenar información la cual será comprimida previamente a ser almacenada, esta compresión deberá realizarse con algoritmos que gestionen flujos de datos, para que no se necesite el fichero total para poder descomprimir la información.

ARQ-06. Permitir ficheros de gran tamaño.

Para poder ser eficaz el sistema de fichero y poderse comparar con otros sistemas de ficheros distribuibles como HDFS, RogerFS debe ser capaz de almacenar ficheros de tamaño masivo del orden de terabytes. Para ello se deberá crear una estructura que soporte estos tamaños con un rendimiento razonable.

ARQ-07. División de los ficheros en bloques distribuidos.

La información se dividirá en bloques, con una forma similar a los bloques de HDFS, estos bloques serán distribuidos por el cluster de la base distribuida que RogerFS tome como base, el tamaño de los

bloques, aunque configurable, será los suficientemente grande para gestionar ficheros de gran tamaño.

3.2 Requisitos del cliente

En esta sección se hablará de los requisitos de la interfaz del cliente y como el usuario podrá utilizar la información almacenada en RogerFS. Para ello, la información se podrá consumir de dos formas:

- **Secuencial.** La información se leerá utilizando comandos similares a otros sistemas de ficheros, y tendrá en cuenta el orden de la información y cómo recuperar esta información para que se puedan utilizar ciertos comandos de consola desde la aplicación cliente, teniendo en cuenta el tamaño de los ficheros.
- **Distribuida.** En este caso, se trabajara con un API compatible con frameworks de procesamiento distribuido, particularmente este API será compatible con el framework Apcache Spark y dará una pequeña librería que permitirá cargar fácilmente RDD desde los ficheros almacenados en el sistema. A la vez se darán mecanismos para poder escribir la información de salida de los trabajos en el propio sistema de ficheros.

Estas dos APIs pondrán a disposición del usuario toda la funcionalidad del sistema de ficheros, y permitirán a los usuarios poder trabajar con ellas; a continuación, se detallan los requisitos de cada una de ellas.

3.2.1 Interfaz secuencial

Esta interfaz estará disponible a través de un API y a través de la propia línea de comandos, estos comandos permitirán al usuario trabajar con el sistema de ficheros de forma muy similar a otros sistemas de ficheros, y a la vez, utilizar comandos comunes usado desde la línea de comando para trabajar con los ficheros.

El objetivo de estos comandos es dar al usuario un acceso exploratorio al sistema de ficheros, que le permita conocer el contenido y la distribución

de los datos. Este acceso no aprovecha la distribución de los datos para ser procesados por lo que no es un método optimo para realizar procesamiento pesado de la información.

Los métodos que están disponibles en la aplicación son similares a los que nos encontraremos en una consola Linux. Todos ellos para ser ejecutados deben empezar con "roger-sh" seguido del comando que deseemos ejecutar. En la siguiente tabla (véase Table 4) enumeramos los comandos utilizados.

Table 4. Lista de comandos secuenciales para RogerFS.

Comando	Parámetros	Descripción
ls	<Directorio>	Devuelve el listado de ficheros y directorios que se encuentra en la carpeta indicada.
cat	<Fichero o Directorio>	Devuelve el contenido de un fichero o un directorio a través de la salida estándar
head	<Líneas> <Fichero o Directorio>	Lee las primeras n líneas de un fichero o un directorio seleccionado y la muestra a través de la salida estándar de la consola
mv	<Fichero Origen> <Fichero o directorio destino>	Mueve un fichero en el sistema de ficheros RogerFS a otra carpeta dentro del sistema de ficheros
upload	<Fichero local> <Directorio o Fichero RogerFS>	Sube un fichero local al sistema de ficheros distribuido, a un directorio concreto
mount	<Fichero de configuración>	Establece un fichero de configuración como base para la configuración de RogerFS, se valida que la configuración sea correcta

Estos comando permiten la gestión básica desde consola del sistema de ficheros y hace posible realizar tareas exploratoria en el sistema de ficheros.

3.2.2 Interfaz de frameworks de procesamiento distribuido

Esta interfaz estará compuesta por un API que permitirá al usuario lanzar tareas usando la frameworks de computación distribuida, en este caso Apache Spark.

Para poder trabajar con Spark se necesita que la información este disponible en un RDD[26]. Para poder crear un RDD se debe definir una colección dentro de los ficheros seleccionados. Para ello, se ha decidido que la interfaz tome el salto de línea como elemento separador, aunque existe la posibilidad de indicar otros elementos.

Una vez indicado como se quiere dividir el fichero, el sistema empieza a trabajar con RDD por lo que las operaciones son similares al trabajo con cualquier otros sistema.

También existe la posibilidad de guardar la salida de un RDD al sistema de ficheros, el sistema almacenara un RDD de Strings como un fichero donde cada línea es un elemento de la colección. En la siguiente tabla (Veasé Table 5) podemos ver los principales método con los que cuenta la interfaz distribuida.

Table 5. Metodos de la interfaz distribuida.

Comando	Parámetros	Descripción
readTextFile	<Fichero o Directorio>	Lee el fichero o el directorio con formato UTF8 y utiliza como separador

[26] El RDD es la interfaz básica con la que cuenta Spark para poder trabajar con la información, en ella se almacena un grafo que permite distribuir los procesos dentro de un cluster, mientras que al usuario se le da una interfaz muy similar a la usada en el manejo de colecciones (Página 18).

		el final de línea
readFile	<Fichero o Directorio> <REGEXP separador> <Codificación>	Devuelve el contenido de un fichero o un directorio a través de la salida estándar
writeFile	<Fichero> <RDD de String> <Separador>	Escribe en un fichero de RogerFS la colección de datos RDD utilizando el separador indicado

Con estos métodos se da acceso al sistema de procesamiento distribuido al sistema de ficheros RogerFS, permitiendo la lectura y escritura de los ficheros en el sistema.

3.2.3 Resumen de requisitos

En esta sección se hará un pequeño resumen de los requisitos obtenidos de las especificaciones de las interfaces del cliente. Estos requisitos deberán validarse por el diseño final.

CLI-01. Interfaz secuencial.

El proyecto deberá contar con un interfaz secuencial que pueda ser llamada desde la propia línea de comandos. Esta interfaz permitirá el análisis de los ficheros secuencialmente y podrá ser combinada con otros comandos de análisis disponibles desde la consola.

CLI-02. La interfaz secuencial contara con comandos exploratorios.

El principal objetivo de la interfaz secuencial es explorar de una forma fácil la estructura y el contenido de los ficheros almacenados en RogerFS, para lo cual se darán comandos que permitan hacer ese análisis como ls, head o cat.

CLI-03. Se mantendrá un orden fijo en todas la operaciones.

Las operaciones secuenciales devolverán las partes de los ficheros siempre en el mismo orden respetando de esta forma la salida de los datos.

CLI-04. Se permitirá la subida de ficheros del sistema local.

Desde la interfaz secuencial se podrán subir ficheros del sistema local al sistema RogerFS, de esta forma será sencilla la inclusión de logs o ficheros que se quieran procesar de forma distribuida.

CLI-05. El sistema debe soportar ficheros de gran tamaño.

Los comandos de la interfaz secuencial deben estar preparados para gestionar ficheros de gran tamaño, por lo que no debe haber limitaciones a la hora de leer estos ficheros o mostrarlo por pantalla.

CLI-06. Interfaz distribuida.

El proyecto incluirá una interfaz distribuida, que permitirá a los sistemas de procesamiento distribuido leer y escribir información en el sistema de ficheros, teniendo en cuenta la localidad de los datos.

CLI-07. La interfaz distribuida debe ser compatible con Apache Spark.

La interfaz distribuida debe ser compatible con el sistema de procesamiento distribuido Apache Spark, y deberá proporcionar métodos para crear RDD de la información contenida en los ficheros almacenados en el sistema de ficheros.

CLI-08. Métodos de lectura de ficheros de texto.

La interfaz distribuida contará con métodos que conviertan ficheros de texto en RDD, en donde los elementos de la colección serán cada una de las líneas contenidas en los ficheros distribuidos.

CLI-09. Métodos de lectura con separadores propios.

La interfaz contará con un método más genérico que permita indicar el separador utilizado para cada uno de los elementos de la colección RDD y la codificación utilizada para poder leer el contenido de los ficheros.

CLI-10. Métodos de escritura de los RDD en el sistema de ficheros.

La interfaz contará con métodos que permitan almacenar la información contenida en un RDD en el propio sistema de ficheros, para poderse después incluir en otros flujos de trabajo más complejos.

3.3 Requisitos de calidad

En esta sección se enumerarán los requisitos de calidad requeridos por el proyecto relacionados con la calidad de código, las pruebas funcionales y la pruebas de disponibilidad y tolerancia a fallos. Estos requisitos aunque no sean funcionales garantizan el correcto funcionamiento del producto en diversas circunstancias.

QAT-01. Guía de estilo de código.

El código deberá cubrir las guías de estilo básica diseñadas para cada uno de los lenguajes utilizados, en este caso al estar desarrollado en Scala, se deberá seguir la guía de estilo de Scala Style (Scala-Lang, 2015), en el caso del código escrito en Java se debe respetar la guía de Google Java Style (Google, 2015).

QAT-02. Test unitarios y porcentaje de cobertura.

El código deberá estar lo suficiente probado garantizando una cobertura de código de al menos el 70%, de esta forma los test garantizarán que cualquier cambio realizado sobre los métodos del proyecto no afecten al código implementado.

QAT-03. Test de integración.

Las pruebas que impliquen conectividad hacia otros sistema como puede ser hacia una base de datos, deberán ser incluidas como test de integración, su cobertura se tendrá en cuenta en el porcentaje de cobertura cubierto por el proyecto.

QAT-04. Test funcionales.

Los requisitos funcionales que han sido expuestos en este documento deben tener asociados al menos un test funcional que garantice su validez. Estos test pueden ser automáticos o manuales, pero en cualquier caso debe existir un procedimiento para ejecutarlos y comprobar su perfecto funcionamiento.

QAT-05. Test de stress y de tolerancia a fallos.

Se plantearan diferentes entornos de test para poder probar el rendimiento y la respuesta del sistema ante caídas. Estos entornos estarán definido y se analizaran los resultados de salida.

3.4 Requisitos de rendimiento

Para poder garantizar la eficacia del sistema no sólo se tiene que cumplir una serie de requisitos de funcionales de calidad, sino que el sistema debe ser eficaz y ser una alternativa real a otras soluciones existente en el mercado por lo que se establecerán unos requisitos de rendimiento para poder medir esta validez tecnológica. Esto requisitos medirán la velocidad del sistema, las capacidades de escalabilidad y las capacidades del almacenamiento.

REN-01. Rendimiento medio del sistema.

Al ser un sistema de ficheros distribuido montado encima de una base de datos, se presupone que la capacidad del sistema debe ser inferior a un sistema de ficheros como HDFS, sin embargo para validar el sistema esta perdida de capacidad no debería superar el 50%.

REN-02. Escalabilidad lineal del sistema.

Debido a las características internas del desarrollo del proyecto y a las capacidades de la base de datos distribuida elegida, el sistema debería ser capaz de escalar al mismo ritmo que escala la base de datos donde la información está almacenada. Por ello, si la base de datos es Cassandra, su escalabilidad es prácticamente lineal, el sistema debería compartir esta característica.

REN-03. Capacidad de almacenar grandes ficheros en el sistema.

El sistema debe contar con la capacidad de almacenar y procesar ficheros de gran tamaños sin perder su rendimiento. El orden de estos ficheros deberá ser de cientos de gigabytes.

3.5 Resumen de requisitos

En la siguiente tabla (véase Table 6) se resumirán todos los requisitos capturados y su prioridad en el proyecto.

Table 6. Resumén de los requisitos del proyecto.

#ID	Nombre	Prioridad
ARQ-01	Sistema de ficheros basado en una Base de Datos distribuida.	Alta
ARQ-02	Minimizar el número de tecnologías necesarias.	Alta
ARQ-03	Ser compatible con sistemas de procesamiento distribuido.	Alta
ARQ-04	Almacenar cualquier tipo de fichero.	Alta
ARQ-05	Permitir compresión en los ficheros.	Baja
ARQ-06	Permitir ficheros de gran tamaño.	Alta
ARQ-07	División de los ficheros en bloques distribuidos.	Alta
CLI-01	Interfaz secuencial.	Media
CLI-02	La interfaz secuencial contara con comandos exploratorios.	Media

CLI-03	Se mantendrá un orden fijo en todas la operaciones.	Alta
CLI-04	Se permitirá la subida de ficheros del sistema local.	Alta
CLI-05	El tamaño de los ficheros no debe limitar la interfaz.	Alta
CLI-06	Interfaz distribuida.	Alta
CLI-07	La interfaz distribuida debe ser compatible con Apache Spark.	Alta
CLI-08	Métodos de lectura de ficheros de texto.	Alta
CLI-09	Métodos de lectura con separadores propios.	Media
CLI-10	Métodos de escritura de los RDD en el sistema de ficheros.	Media
QAT-01	Guía de estilo de código.	Alta
QAT-02	Test unitarios y porcentaje de cobertura.	Alta
QAT-03	Test de integración.	Media
QAT-04	Test funcionales.	Baja
QAT-05	Test de stress y de tolerancia a fallos.	Baja
REN-01	Rendimiento medio del sistema.	Media
REN-02	Escalabilidad lineal del sistema.	Alta

REN-03.	Capacidad de almacenar grandes ficheros en el sistema.	Media

En el capítulo de evaluación y pruebas (véase página 71) se hará un análisis de todos los requisitos que ha cumplido el proyecto y que requisitos se ha quedado fuera del alcance o han resultado imposibles de satisfacer.

4 Diseño

En esta sección se hablará del diseño propuesto para el proyecto, de los módulos que contiene y del modelo de datos que utiliza el sistema para almacenar la información en la base de datos. El objetivo de esta sección es dar una visión del estado general en el que se encuentra el proyecto, y analizar en detalle cada una de las partes del mismo.

El proyecto se ha desarrollado en Scala (Typesafe, 2015) y Java, y se ha utilizado el gestor de proyectos SBT (SBT, 2015), este gestor de proyecto tiene dos funcionalidades básicas: gestionar las dependencias del proyecto y crear proyecto multi-modulo. En este caso el proyecto se ha estructurado utilizando estas dos características. Por ello, se ha creado un proyecto divido en módulos para dar soporte a todos los requisitos del proyecto, En la figura, que se presenta mas adelante,(Véase Figure 7) se puede ver la jerarquía que tiene cada uno de los siguientes módulos:

- **Common.** Contiene las clases e interfaces que son comunes a los demás proyectos.
- **Core.** Contiene toda la funcionalidad básica del proyecto así como las librerías necesarias para trabajar con el sistema de ficheros RogerFS.
- **Cassandra Store.** Contiene la información necesaria para poder almacenar y buscar información usando Cassandra como base de datos distribuida.
- **Shell.** Este módulo implementa los comandos que pueden ser ejecutados desde la consola para poder trabajar con la interfaz secuencial de RogerFS.
- **Distribution.** Contiene el script que crea el TARBALL que se puede distribuir a los usuarios.
- **Test.** Contiene un Store en memoria que permite probar todas las funcionalidades del sistema, y crear test unitarios sin necesidad de tener una base de datos conectada.

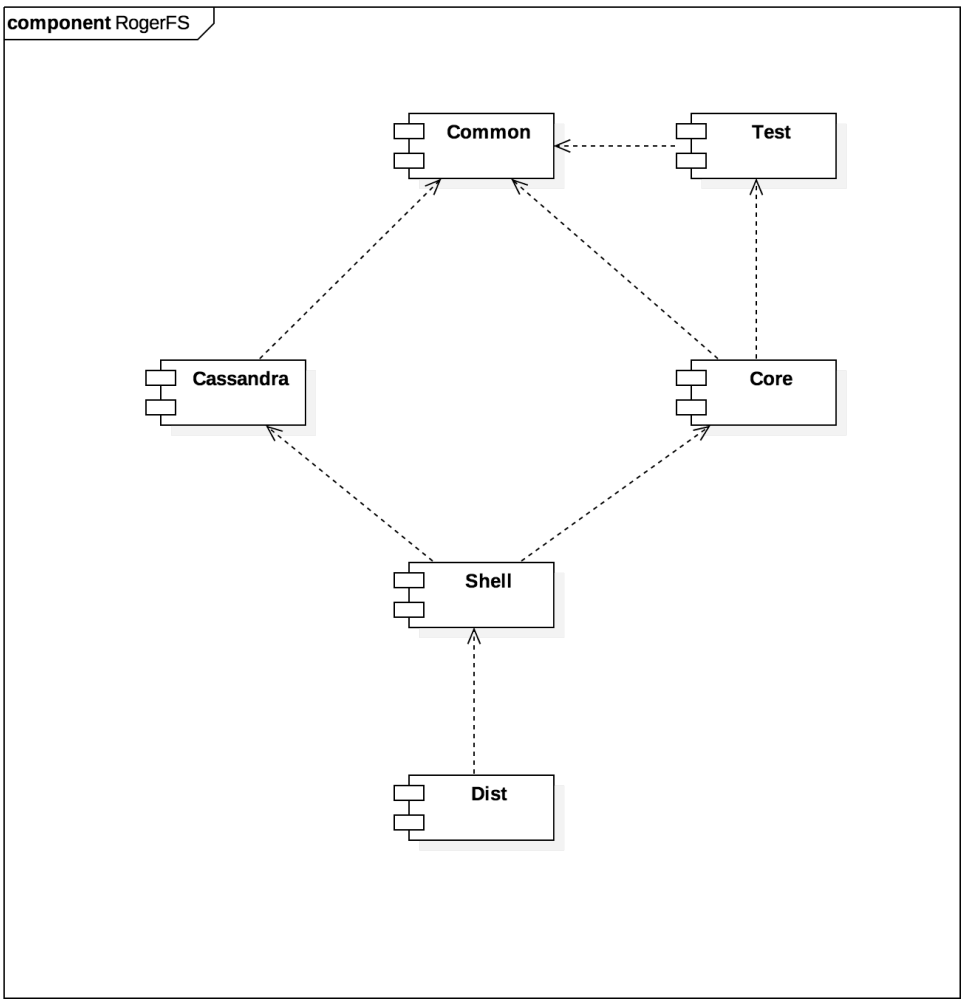

Figure 7. Jerarquía de módulos en RogerFS.

Esta arquitectura, basada en módulos, permite tener una gran extensibilidad en el sistema, aunque actualmente existe una implementación sólo para Cassandra es muy fácil extender el sistema para soportar otras bases de datos, utilizando los mismos comandos. Este trabajo de extensibilidad se ha realizado utilizando interfaces que permitan trabajar con estos comandos sin tener en cuenta la tecnología que se está usando.

Teniendo en cuenta esto, se desarrolló un módulo de test que cuenta con una implementación de el Store de RogerFS que permite trabajar con todas las funcionalidades de éste sin tener en cuenta la base de datos con la que tengamos que trabajar.

El modulo de Cassandra Store implementa la funcionalidad necesaria para trabajar con la información almacenada en la base de datos.En la siguiente imagen (

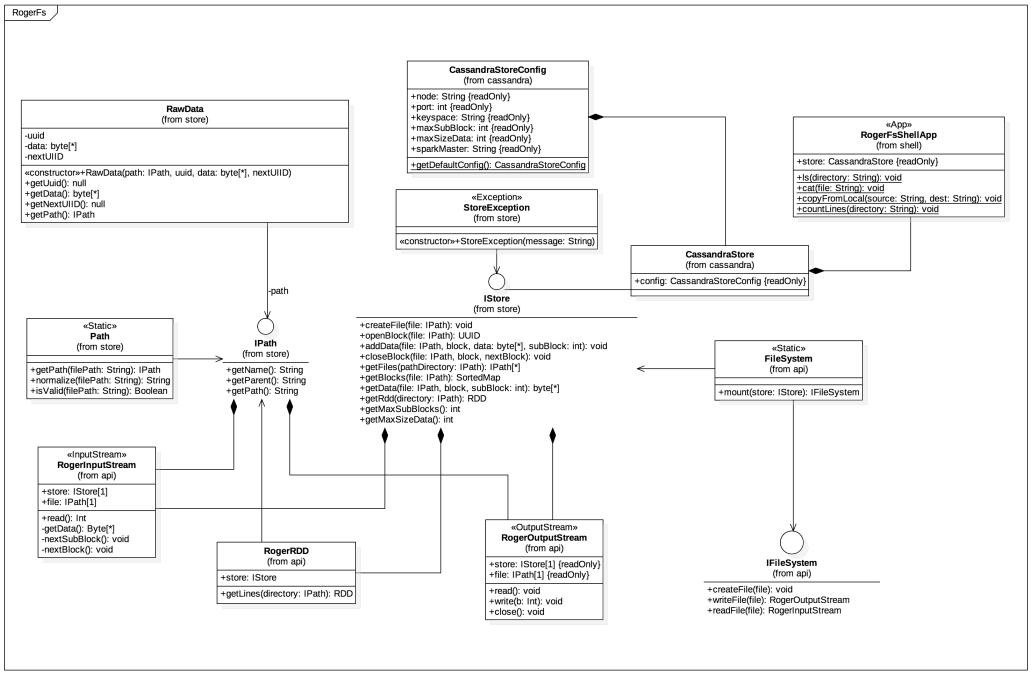

Figure 8), se muestra una imagen con el diagrama de clases desarrollado para RogerFS, en los próximos apartados no centraremos en cada uno de los módulos implementados.

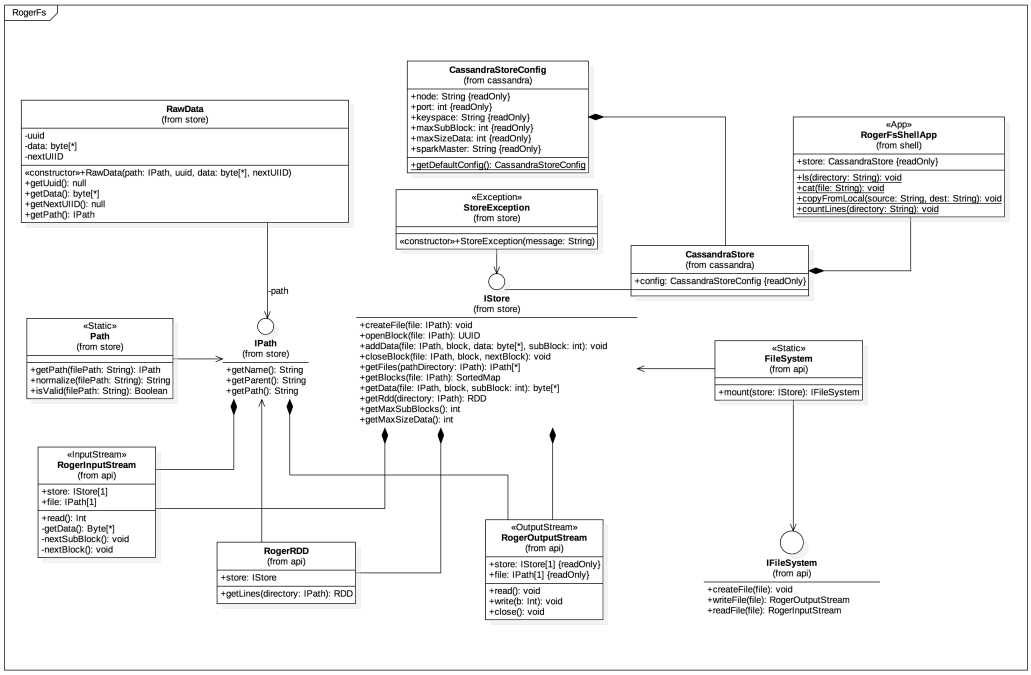

Figure 8. Diagrama de clases de RogerFS

4.1 Módulo Common

Este módulo almacena toda la información que es compartida entre los diferentes proyectos. De esta forma se puede crear una capa estanca que permita la interoperabilidad sin tener que incurrir en dependencias cíclicas que pondrían en riesgo la integridad del sistema.

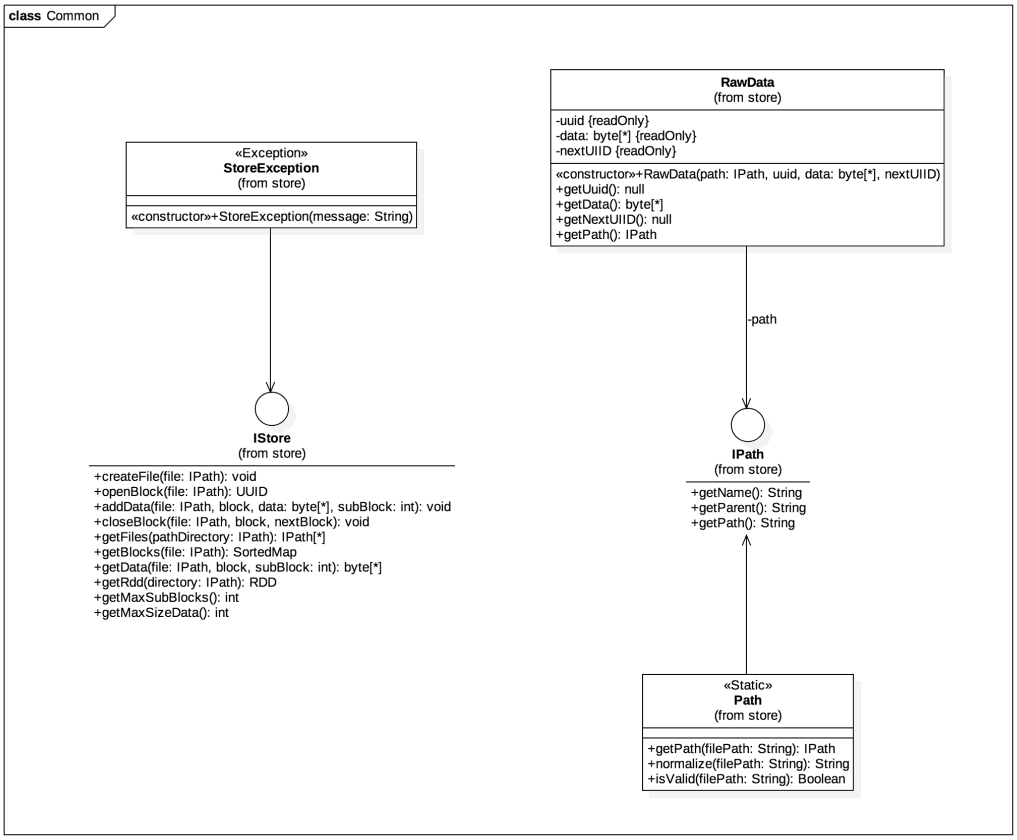

Figure 9. Detalle de las clases en el módulo de common.

En la imagen anterior (Figure 9) se ven la clases e interfaces que están incluidas en el módulo. La clase principal es la interface IStore que permite desarrollar módulos que conecten el sistema de fichero con otras bases de datos.

4.1.1 Interfaz IPath

Interfaz que permite devolver información de las rutas internas del sistema de ficheros. El sistema maneja esta interfaz para trabajar con las rutas internas y garantizar que los objetos tienen la validación correcta. El Factory de esta interfaz es Path, el cual realiza todas las comprobaciones para validar que las rutas son correctas.

- **getName(): String.** Este método indica el nombre del fichero final, en una ruta UNIX "/abc/def/ghi.jkl" , devolviendo el nombre del fichero con el formato "ghi.jkl".
- **getParent(): String.** Este método indica el nombre del directorio padre, en una ruta UNIX "/abc/def/ghi.jkl" , devolviendo una ruta con el formato "/abc/def/".
- **getName(): String.** Este método indica la ruta completa al fichero referenciado, esta ruta debe ser única y estar construida con la información de los dos métodos anteriores, en el caso de una ruta UNIX ("/abc/def/ghi.jkl"), devolviendo una ruta completa con el formato "/abc/def/ghi.jkl".

Existe una clase interna llamada *InternalPath* que implementa la interfaz y permite instanciar la clase. Esta clase implementa la interfaz considerando que las rutas que se usan en el sistema de ficheros son similares a las ruta utilizadas en cualquier sistema UNIX, sin embargo, el sistema en si no depende de estas características y es capaz de implementar otros tipos de rutas o organización de la información siempre que se cumplan los siguientes requisitos.

- La organización es jerárquica, dos hermanos deben devolver la mima cadena padre.
- Para considerar un elemento diferente toda la ruta debe ser diferente y inequívoca.
- Con los métodos *getParent* y *getName* se debe poder completar el método *getPath*.

4.1.2 Interfaz IStore

Esta interfaz permite ser independiente de la base de datos que se utilice para almacenar y buscar la información gestionada por RogerFS. La interfaz incluye toda la funcionalidad necesaria para gestionar las tres capas necesaria del sistema de ficheros:

- **La estructura de metadatos.** La creación del árbol de carpetas y ficheros que incluirá el sistema es gestionado con esta interfaz.
- **La escritura y lectura de datos.** Se establecen métodos para poder escribir y leer información del sistema, estos método intentan ser sencillos de gestionar permitiendo al sistema que haga sus optimizaciones internas.
- **Instanciación del RDD.** La interfaz devuelve una estructura RDD que permita trabajar con la base de datos interna utilizando el sistema Spark.

También establece un modelo de datos simple que permite a los usuarios trabajar con los datos del propio sistema.

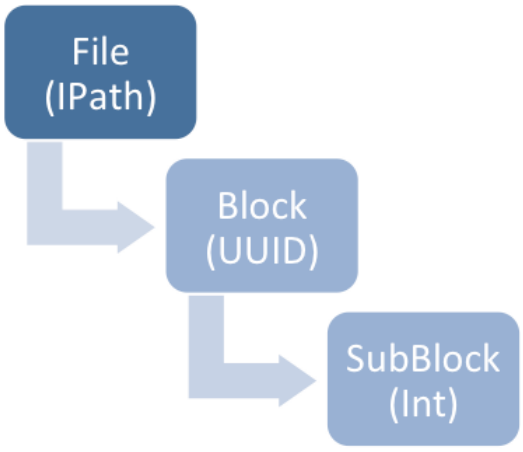

Figure 10. Modelo de datos interno de RogerFs.

En la imagen anterior (Figure 10) podemos ver el modelo de datos utilizado de manera interna por RogerFS. Aunque la implementación final de cada uno de estos elementos depende de la base de datos subyacente, al tratarse de un sistema pensado para trabajar con sistemas distribuidos el modelo contiene cierta relación semántica sobre este tema.

- **File.** Es la unidad mínima que el usuario conoce del sistema de ficheros, cuando el usuario quiere buscar un documento se refiere a él por esta entidad.
- **Block.** Un fichero está dividido en bloques, estos bloques están repartidos por la base de datos distribuida subyacente. El sistema asume que un bloque de información no esta divido en diferentes máquinas, por lo que debe ocupar como máximo el espacio disponible de un solo nodo.
- **SubBlock[27].** Cada bloque esta divido a su vez en diferentes sub-bloques, estos son la unida mínima de acceso del sistema.

A continuación haremos una descripción de cada uno de los métodos que engloban la clase IStore.

- **createFile(file:IPath).** Envía a la base de datos la indicación de que el sistema va a crear un nuevo fichero en el path indicado. El sistema de fichero deber preparar el sistema de datos para este nuevo fichero.
- **openBlock(file:IPath):UUID.** RogerFS indica al sistema de fichero de almacenamiento subyaciente que va crear un nuevo bloque información, este bloque podra recibir datos hasta que se cierre. En el momento de la creación el sistema devolverá el identificador único que permitirá interactuar con él.
- **addData(file:IPath, block:UUID, data:byte[], subBlock:Int).** Una vez abierto el bloque se puede comenzar a escribir en él, para ello se debe indicar el identificador único del bloque que se quiere escribir, el subBlock donde se almacenarán los datos.
- **closeBlock(file:IPath, block:UUID, nextBlock:UUID).** Cuando se decide cerrar un bloque porque se ha alcanzado el tamaño máximo

[27] La creación de la unidad de subloque permite a los sistema subyaciente trabajar de forma más sencilla con la información y reduce la problemática de enviar grandes ficheros a través de la red, reduciendo de esta forma la latencia del sistema.

de este o porque se ha finalizado la escritura de este, se llama a este método para que se cierre el mismo y se indique a partir de donde se empieza a leer el siguiente bloque. En caso de que este bloque sea el último el campo de nexblock se queda como vacío o NULL.

- **getFiles(pathDirectory:IPath):IPath[].** Recupera una lista de ficheros y de directorios que son hijos del path indicado, este método se utiliza para recuperar la información de una carpeta concreta y poder analizar todos los ficheros incluidos a la vez. Es un proceso muy util para el análisis con frameworks de computación distribuida.
- **getBlocks(file:IPath):SortedMap<UUID,UUID>.** Recupera todos los bloques asociados a un fichero concretos de esta forma se puede hacer un filtrado de estos de una forma más sencilla. En caso de que el sistema no pueda procesar la información de forma conjunta el sistema podrá recuperar estos bloque utilizando los identificadores único generados.
- **byte[] getData(file:IPath, block:UUID, subBlock:Int).** Obtiene los datos de un sub-bloque concreto del sistema . Estos datos se devuelven en un byte array en caso de que el subBlock no contenga información devuelve un NULL.
- **int getMaxSubBlocks():Int.** Es el número máximos de subBlock que contiene un bloque concreto del sistema. RogerFS no va a mandar información que supere este número de subBlock concreto.
- **getMaxSizeData():Int.** Es el número de bytes máximos que puede almacenar un subBlock dentro del sistema. RogerFS necesita esta información ya que es sistema subyacente el que conoce cuales son sus limitaciones físicas.
- **getRDD(path:IPath):RDD.** Recupera un RDD con la información contenida en el path enviado, en caso de que el path sea un directorio del sistema el RDD devolverá toda la información de los ficheros almacenados en ese directorio. Esta búsqueda no se realiza recursivamente.

Si se analiza el anterior interfaz se puede observar que para poder escribir información se debe seguir un flujo concreto de acciones, en la siguiente imagen (Figure 11) se describe como es dicho flujo. En el se puede observar el orden de invocación de los métodos openBlock y closeBlock en caso de que se quieran seguir agregando datos en el sistema o cuando se recibe el último bloque de un fichero.

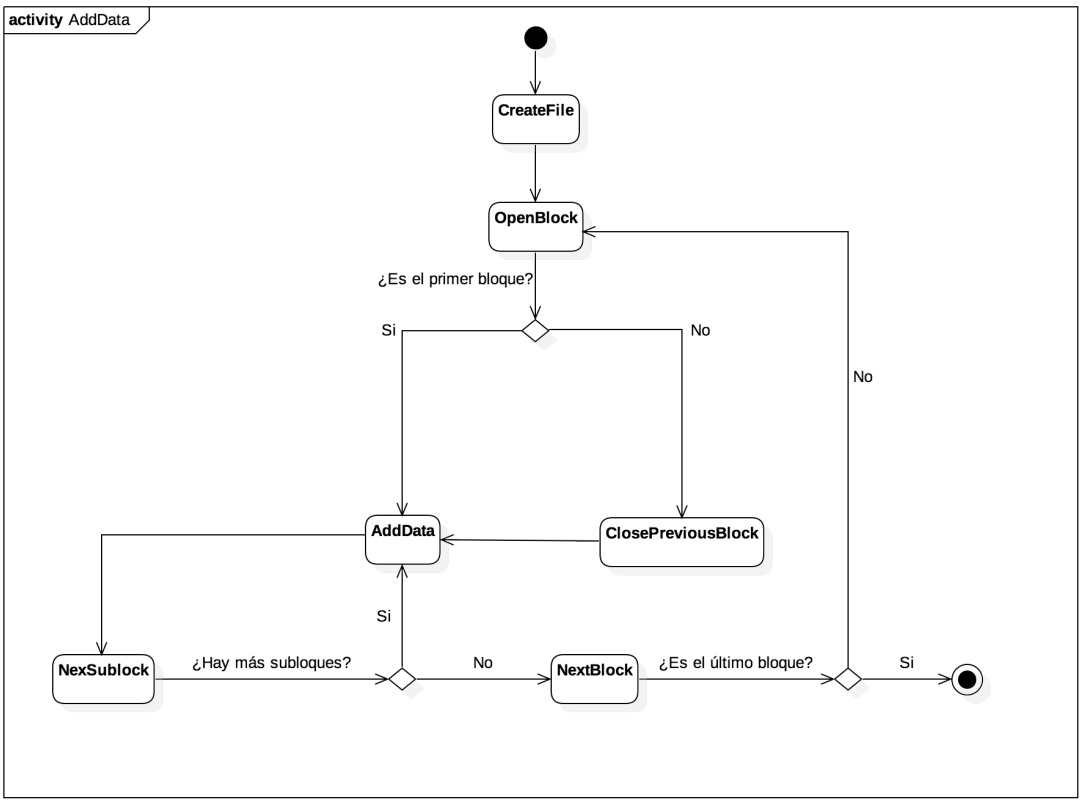

Figure 11. Diagrama de flujo de inserción de datos en RogerFS.

4.2 Módulo Core

El módulo de Core de RogerFS se encargar de toda la lógica del sistema, utilizando la interfaz de IStore como medio de abstracción, permite que no se necesite conocer la base de datos distribuida que se esta utilizando, lo cual permite que RogerFS este preparado para ser utilizado con diferentes sistemas. En la siguiente imagen (Figure 12) se ve un detalle de las clases que componen este módulo.

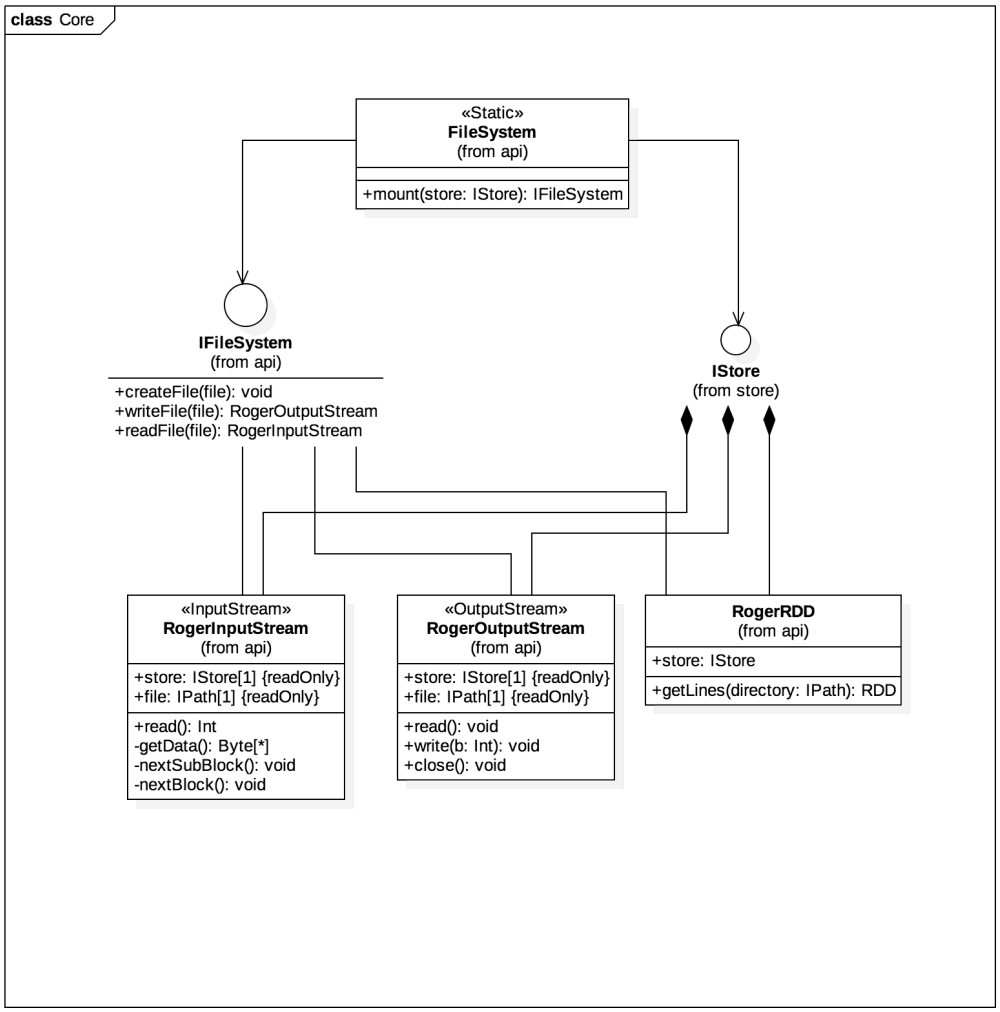

Figure 12. Detalle de las clases en el módulo de Core.

A continuación se hará una descripción de cada una de las clases de este módulo y sus funciones dentro del sistema.

4.2.1 Interfaz IFileSystem

Esta interfaz es la que permite el acceso a la interfaz secuencial descrita en el apartado de requisito (página 43). La creación de esta interfaz facilita el acceso al API, a aplicaciones que no se hayan realizado con escala, evita que ciertas características pudieran estar oculta o no accesibles desde otros lenguajes como por ejemplo Java. Es una buena práctica exponer el

API pública a través de una interfaz para poder de esta forma monitorizar los cambios que se puedan producir.

- **createFile(file:IPath).** Crea un fichero en el sistema para que se pueda escribir en él. Es el primer paso necesario para poder trabajar con ficheros en RogerFS
- **writeFile(file:IPath): RogerOutputStream.** Devuelve un objeto RogerOutputStream que permite al usuario escribir en el sistema de ficheros RogerFS como si estuviera escribiendo en un sistema de ficheros local.
- **readFile(File IPath): RogerInputStream.** Devuelve un objeto RogerInputStream que permite leer al usuario los ficheros de manera secuencial, como si de un fichero en el disco local se tratase.

4.2.2 Clase FileSystem

La clase FileSystem permite montar unidades de RogerFS, para ello solo se tiene que pasar como parámetro el objeto IStore que queremos utilizar, esto permite montar diferentes unidades en diferentes base de datos distribuidas o en la misma base de datos con diferentes configuraciones.

4.2.3 Clase RogerOutputStream

La clase RogerOutputStream hereda de la clase OutputStream del API estándar de Java. Se encarga de escribir ficheros dentro del sistema RogerFS, el usuario no se tiene que preocupar del tamaño de los subBlock, ni del numero de subBlock máximos configurados en la base de datos. Para escribir sigue el flujo que aparece en la imagen anterior (Figure 11).

Esta clase no es thread-safe y requiere de una nueva instancia de objeto por cada hilo que escriba en el disco.

4.2.4 Clase RogerInputStream

La clase RogerInputStream hereda de la clase InputStream del API estándar de Java. Esta permite leer ficheros almacenados en el sistema de

ficheros RogerFS no hace falta conocer la estructura interna de los ficheros en el sistema de almacenamiento. La clase RogerInputStream se encarga que el proceso sea completamente transparente al usuario.

Esta clase no es thread-safe y requiere de una nueva instancia de objeto por cada hilo que lea del disco.

4.2.5 Clase RogerRDD

Permite trabajar con RDD en un sistema de ficheros de RogerFS la información recuperada es tratada para poder trabajar con grandes ficheros de texto. Esta clase permite trabajar con grandes ficheros de textos utilizando tecnologías de procesamiento distribuido como Apache Spark.

El método getLines devuelve un RDD que representa una colección de cadena de texto, cada uno de las cadenas equivale a una línea del fichero original. Sin embargo, para poder realizar esta tarea el sistema se requiere pre-procesar la información almacenada en el sistema de ficheros, debido a que la división por bloques que realiza el sistema no tiene en cuenta el final de las líneas por lo que se necesita reconstruir estos datos antes de devolver la información.

En la siguiente imagen (Figure 13), se puede ver un diagrama que representa este proceso, la salida es un RDD que contiene la información ya procesada para poder trabajar tal como se describió este método.

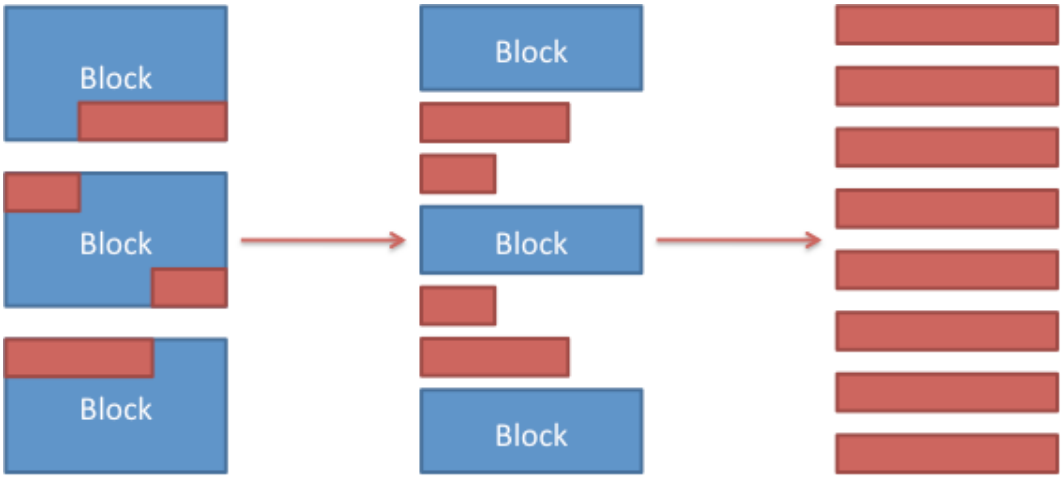

Figure 13. Diagrama de como se recostruyen las lineas a través de los bloques.

La reconstrucción se hace en dos fases, en la primera se recuperan las líneas que están en los limites de los bloques, después se unen estas líneas en una segunda fase y se presentan al usuario sin repeticiones.

4.3 Modulo Shell

El módulo Shell esta encargado de mostrar al usuario las aplicaciones necesarias para trabajar con el sistema de ficheros RogerFS desde la consola, para ello expone una serie de métodos que permite navegar y subir ficheros del sistema local al sistema de ficheros distribuido. En el siguiente diagrama se puede ver la clases implicada en este proceso.

En la siguiente tabla (Table 7) se describe las aplicaciones que se pueden lanzar desde una consola con rogerfs-shell desplegado.

Table 7. Comandos incluidos en el modulo shell.

Comando	Parámetros	Descripción
cat	<ruta_fichero>	Devuelve la información almacenada en RogerFS en formato texto, recorre cada bloque en formato secuencial, si se quiere el siguiente bloque basta con pulsar un tecla

ls	<ruta_fichero>	Devuelve todos los ficheros y directorios que se encuentran en una ruta concreta del sistema. En caso de que la ruta no sea correcta o no exista la aplicación devolverá un mensaje indicando dicha situacio.
mv	<ruta_origen> <ruta_destino>	Permite mover un fichero a una nueva ruta del sistema, para ello se sobrescriben las rutas de todos los ficheros implicados en el proceso.
copyFromLocal	<ruta_origen_local> <ruta_destino>	Se envía un fichero desde una ruta local al sistema de ficheros RogerFS, para ello se utiliza la interfaz secuencial de escritura.

Todos estos comando permiten ser incluidos dentro de otros comandos y permite al usuario trabajar desde el terminal fácilmente con la información almacenada en el sistema de ficheros.

4.4 Modulo CassandraStore

Este módulo es el que da la interoperabilidad entre el sistema de ficheros RogerFS y el la base de datos distribuida Cassandra. Para ello, se ha necesitado que haya una clase que implemente la interfaz IStore y una clase que se encargue de leer la configuración recuperada del sistema que permita montar las unidades requeridas.

En la siguiente figura (Figure 14) se muestran las principales clases que componen este módulo, estas clases tienen toda la funcionalidad necesaria para trabajar con Cassandra desde RogerFS.

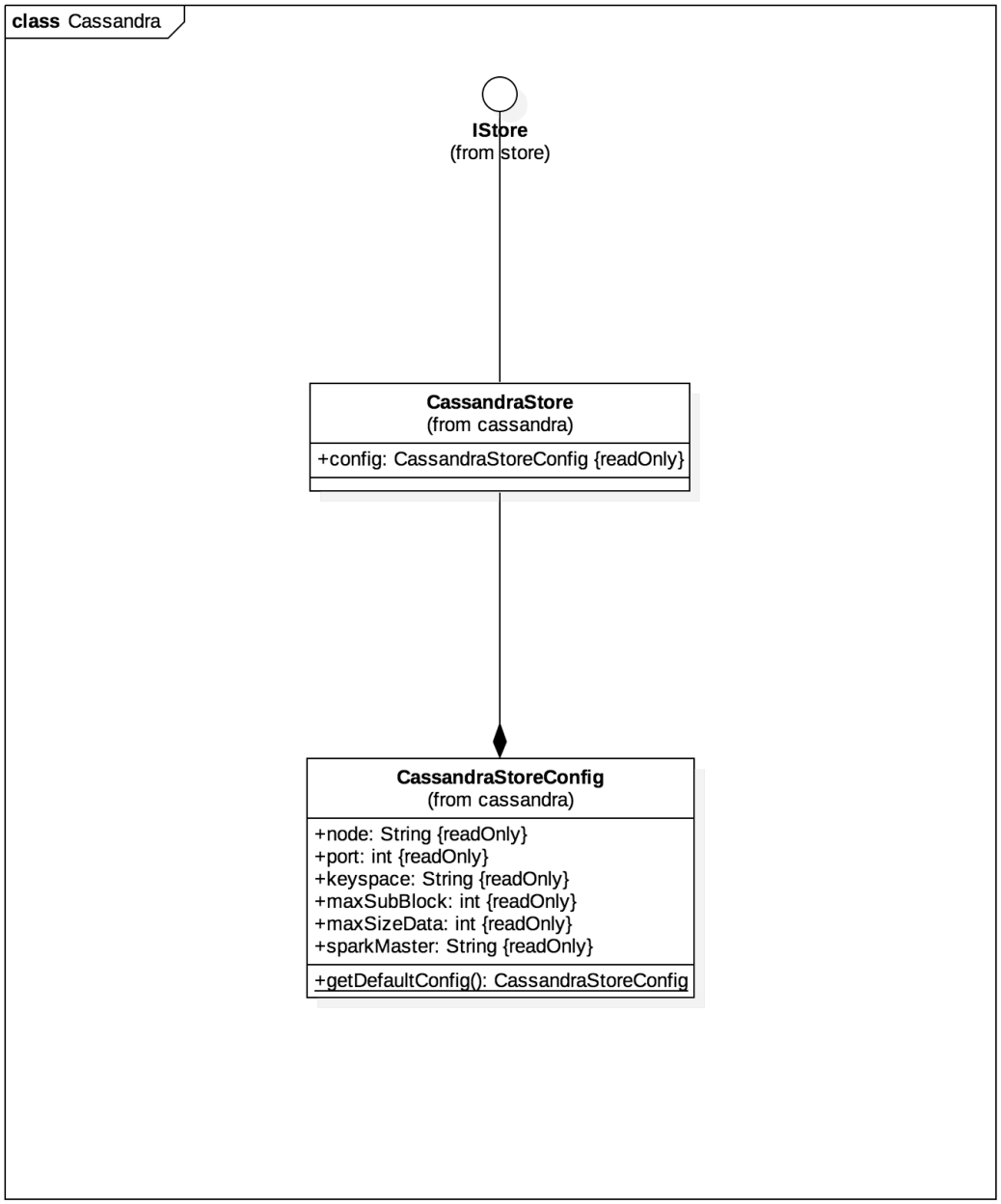

Figure 14. Diagrama de clases implicadas en el módulo CassandraStore.

Para poder almacenar la información necesaria en Cassandra el sistema necesita crear una tabla donde almacenar la estructura de la información y los datos en crudo. En la siguiente tabla (Table 8) se muestra los campos de esta tabla.

Table 8. Descripcion de los campos de CassandraStore.

Campo	Características	Descripción
Path (text)	Primary Key, Partition Key, Indexed	Es el nombre del fichero, esta incluida en la clave primaria del fichero.
Parent (text)	Primary Key, Partition Key, Indexed	Es la ruta del directorio que contiene el fichero, esta información se utiliza para recuperar los datos de los directorios.
Block (UUID)	Primary Key, Partition Key	Identificador del bloque donde se está escribiendo. Un bloque contiene subBlock y estos están almacenados en la misma máquina.
Sublock (Int)	Primary Key, Cluster Key,	Índice del subBlock que se está almacenando, este nunca debe sobre pasar el valor máximo de subBlock indicado por el driver del datastore.
NextBlock (UUID)	static	Guarda la información del siguiente elemento, es null cuando es el último bloque de la cadena.
Data (Blob)		Lugar donde se almacena la información de RogerFS

El diseño de esta tabla utiliza diferentes características de la base de datos subyacente, como el uso de las filas anchas, o el uso de campos estáticos.

- **Filas anchas.** Este es un patrón de diseño típico en base de datos orientadas a columna como Cassandra o HBase. Debido a que el sistema garantiza que una fila no va a dividirse en diferentes máquinas y que la columnas que contiene una fila están ordenadas.

El usuario utiliza las filas para almacenar información relacionada para ser consumida por bloques.

- **Columnas estáticas.** Son columnas que están compartidas por diferentes filas, una modificación en esta columna afecta a todas las filas.

En la siguiente figura (Figure 15) se puede ver una representación de cómo se almacenaría de forma física la información en la base de datos[28].

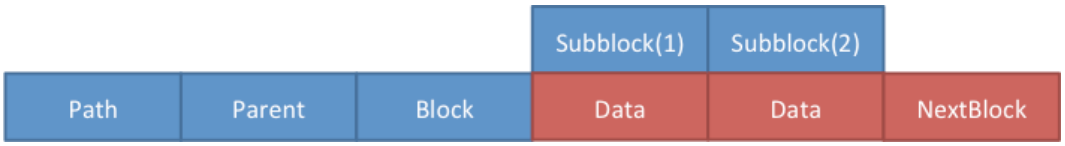

Figure 15. Modelos de datos fisico en Cassandra.

Al ser la cable de cluster SubBlock cada nuevo elemento permitirá almacenar información en la misma fila que se utiliza. Por otro lado, al ser NextBlock una columna estática cualquier cambio que se realice en esta columna afecta a los datos del resto de filas. Tanto la información contenida en el Path como en el Parent, están indexadas de esta forma permitimos al usuario poder hacer búsquedas por estos campos. Este modelo de datos nos permite escalar hasta los terabytes, manteniendo las capacidades distribuidas que ofrece la base de datos Cassandra.

4.5 Modulo Test

Para poder realizar la pruebas del sistema sin tener que levantar un servidor de Cassandra, se pensó desde en principio en los test que se iban a lanzar sobre el código desarrollado. Por ello, toda la funcionalidad relacionada con la propia base de datos está agrupada en un solo módulo. Gracias a este encapsulamiento se han conseguido dos ventajas: la primera la posibilidad de poder integrar RogerFS en el futuro con otras base de datos, la segunda, la posibilidad de poder crear una clase *mockup* que nos

[28] Esta forma de modelar datos es muy común cuando se utilizan bases de datos orientadas a columnas, se pueden ver más ejemplos de estas arquitecturas en el siguiente enlace (https://youtu.be/qphhxujn5Es)

permita hacer test unitarios sin tener que tener una base de datos por debajo del sistema.

El módulo de Test provee una clase básica que implementa un IStore en memoria que nos permite validar el funcionamiento del sistema completo, y poder hacer test unitarios sobre él. Esta implementación es sencilla y permite trabajar sin tener en cuenta el datastore que se encuentra por debajo.

5 Evaluación y pruebas

En esta sección analizaremos los resultados de una comparativa que se realizo al sistema en diversos escenarios.

El desarrollo de estas pruebas se han hecho utilizando el servicio Amazon AWS, el cual nos permite desplegar y gestionar servicios en la nube. En el primer apartado de esta sección se describirá la topología desplegada y en los diferentes escenarios.

Después se describirá la comparativa que se ha realizado en este entorno. Esta comparativa está divida en dos partes: la primera orientada a validar la escalabilidad del sistema tanto en escrituras como en lecturas; y la segunda, pensada para comparar el sistema RogerFS con un sistema con similares características, en este caso HDFS.

5.1 Topología de pruebas

En esta sección se describirá el despliegue utilizado para las pruebas de rendimiento que se han realizado al sistema, en primer lugar se indicará cual ha sido el hardware seleccionado para las pruebas, y después se explicarán las arquitecturas de despliegue de los sistemas comparados, en este caso dos: RogerFS con Cassandra y Spark y Spark con HDFS.

5.1.1 Hardware seleccionado

Para el desarrollos de la pruebas se ha utilizado los servicios de Amazon AWS, más concretamente el servicio de Amazon EC2 (*Elastic Cloud Computing*), este servicio es un IaaS (*Insfrastructure as a Service*), que permite provisionar la infraestructura necesaria para poder desplegar aplicaciones escalables en muy poco tiempo. Para ello sólo es necesario seleccionar el tipo de instancia que se quiere desplegar y la imagen con la que se va arrancar la instancia. El servicio permite añadir todas las instancias que se deseen y el coste de las mismas se tarifica según el tiempo de uso de las mismas.

Para la pruebas que se han realizado se han desplegado en instancias Amazon m3.xlarge, en la siguiente tabla (Table 9) se puede ver un detalle de las principales características de este tipo de instancia.

Table 9. Caracteristicas de las instancias Amazon m3.xlarge.

Caracteristica	Valor	Comentarios
vCPU	4	Procesadores Intel Xeon E5-2670 v2 (Ivy Bridge) de alta frecuencia.
Memoria	15GiB	--
Almacenamiento	2x40GiB	Almacenamiento de instancias basado en SSD para un rápido rendimiento de E/S.
Redimiento de Red	Alto	Red bajo demanda.

La selección de este esta tipo de instancia para el desarrollo de la pruebas se debe al equilibrio de la misma en tres factores, memoria, tipo de almacenamiento y calidad de red.

- **Memoria.** El sistema RogerFS es un sistema pensado para ser usado por sistemas de procesamiento modernos como Apache Spark, esto sistemas hacen uso de la memoria para poder procesar la información almacenada en el sistema, por tanto la instancia elegida debería tener suficiente memoria para que el procesamiento de grande cantidades de información fuera posible.
- **Almacenamiento.** Las base de datos como Apache Cassandra intentan aprovechar al máximos sistemas de almacenamiento,

como se puede leer en la documentación de Cassandra[29], los discos duros de estado solido son recomendados, debido a que poseen un baja latencia en lecturas aleatorias y son capaces de dar una muy buenas prestaciones en escrituras secuenciales utilizando mecanismos de compactación. Debido a esto la instancia seleccionada debía contar con discos duros de estado solido.

- **Red.** En los sistemas distribuidos la red es uno de los mayores limitantes a la hora de trabajar en "Las ocho falacias de la computación distribuida" (Deutsch, 1995), queda claro que la red juega un papel muy importante en cualquier sistema distribuido y especialmente en un sistema masivo de almacenamiento debido a que el tráfico de red será sustancialmente mayor que en otros sistemas. Por ese motivo, la instancia seleccionada debía tener una red de altas prestaciones que permitiera tener un flujo constante de información disponible.

Por todas esta razones, la instancia seleccionada fue la m3.xlarge, ya que cumplía todos estos factores, manteniendo un buen equilibrio calidad/coste.

5.1.2 Despliegue de RogerFS con Cassandra

Para realizar las pruebas de RogerFS, el sistema necesita dos componentes principales la base de datos Apache Cassandra y el sistema de procesamientos distribuido Apache Spark. Para la instalación de estos componentes se optó por un despliegue clásico. En el cual, cada uno de los nodos de datos de Cassandra tendría un nodo de trabajo de Spark. Debido a que Spark tiene la capacidad de seleccionar los nodos de trabajo teniendo en cuenta la localidad de los datos, esta configuración permite a los nodos de trabajo de Spark, aprovechar al máximo los recursos de las máquinas y no tener que pasar por la red para recoger los datos de cada

[29] http://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architecturePlanningHardware_c.html

unos de los nodos de Cassandra, optimizando el consumo de ancho de banda y las latencias que se pueden producir en el sistema.

La configuración de los nodos de Cassandra será la misma en cada una de las pruebas, solamente se modificará el factor de replicación del Keyspace, el cual indica que los nodos deben contener la misma información para proveer al sistema de tolerancia a fallos; y por otro lado el modo de consulta el cual se utilizara o ONE o QUORUM. El modo de consulta se indica en el cliente y garantiza recupera la información con consistencia.

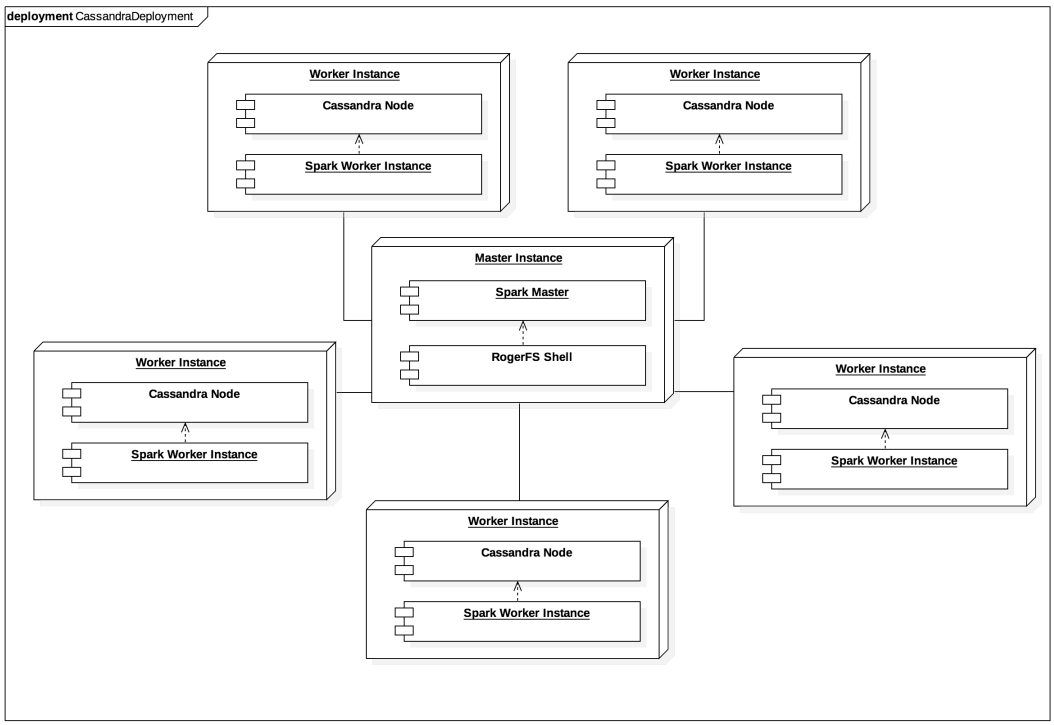

Figure 16. Diagrama del despliegue del sistema RogerFS con Cassandra con 5 nodos.

En la anterior figura (Figure 16) se describe el despliegue realizado para el sistema cuando este cuenta con 5 nodos.

Como se puede observar además de contar con los cinco nodos de trabajo, se incluye un nodo maestro que contendrá dos componentes, el maestro de Spark, el cual coordina las ejecuciones del sistema de procesamiento distribuido, y por otro lado, el cliente de RogerFS, el cual se encargará de las inserciones en el sistema y de la comunicación con el sistema de fichero distribuido.

La configuración de Apache Cassandra y de Apache Spark es la que viene por defecto, sólo se ha incluido en el caso de Apache Spark el Spark Cassandra Driver, el cual permite al sistema de procesamiento distribuido ser capaz de escribir y de leer de Cassandra.

En la siguiente tabla (Table 10) se puede ver un listado del software utilizado para el despliegue de RogerFS y los sitios de donde se obtuvo los distribuibles.

Table 10. Versiones del software utilizado en el despliegue de RogerFS.

Tecnología	Versión	URL
Apache Cassandra	2.1.8	http://www.planetcassandra.org/
Apache Spark	1.4.1	http://spark.apache.org/
Cassandra Spark Connector	1.4	https://github.com/datastax/spark-cassandra-connector

5.1.3 Despliegue de Spark con HDFS

Para poder compara el sistema de ficheros RogerFS con otro sistema se decidió utilizar otro sistema con similares características y con la suficiente madurez y presencia en el mercado, HDFS. Para el sistema de procesamiento distribuido se opto por el mismo seleccionado para RogerFS de esta forma se podrán comparar los dos sistema en el mismo escenario.

Para la configuración del sistema se utilizo una instalación clásica, en donde los nodo de datos de HDFS están desplegados en las misma instancias que los trabajadores de Spark. De esta forma Apache Spark puede aprovechar su capacidad para seleccionar los nodos basándose en

la localidad para recuperar la información de cada uno de los nodos de datos.

La configuración de ambos sistemas es la que viene por defecto, modificando sólo aquellos parámetros necesarios para el corrector funcionamiento en el sistema.

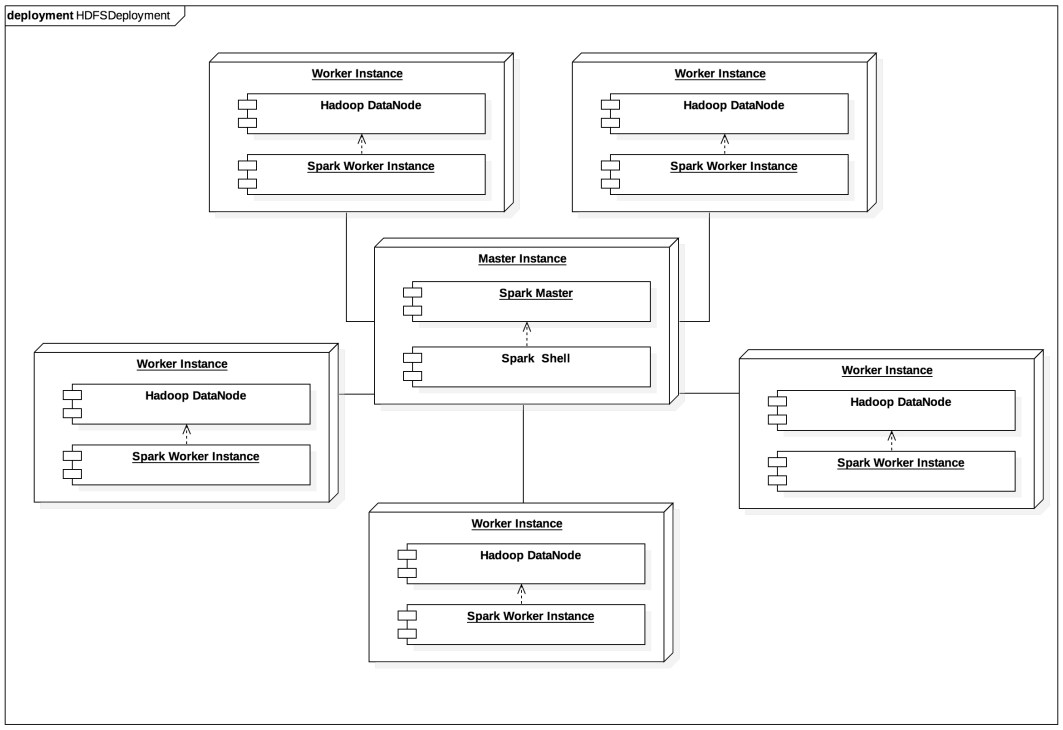

Figure 17. Diagrama del despliegue de HDFS con Spark en 5 nodos.

En este caso, como en el anterior el despliegue de Spark utilizando el modos sin gestor de cluster, debido a que las latencias introducidas por este sistema podrían falsear los resultados, y las funcionalidades que proporcionan no son necesarias para este escenario.

Los único parámetros que se modifican en los test son el factor de replicación, el cual indica el numero de data nodes que tienen un mismo dato, el resto de los parámetros se mantienen en todos las pruebas.

En la siguiente tabla (Table 11) se muestra la versiones y los enlaces donde se han obtenido los distribuibles.

Table 11. Versiones del software utilizado en el despliegue de HDFS.

Tecnología	Versión	URL
Apache HDFS	2.7.1	http://hadoop.apache.org/docs/stable/
Apache Spark	1.4.1	http://spark.apache.org/

5.2 Evaluación de escalabilidad del sistema

Siendo un sistema distribuido orientado al procesamiento de grandes volúmenes de información, la escalabilidad horizontal es un requisito necesario para el sistema.

El objetivo de esta prueba es evaluar la escalabilidad del sistema, para ello se dispondrá de el mismo set de datos de información y se evaluaran tanto la velocidad de operaciones de escritura como de lectura. También se analizará como afecta los diferentes factores de replicación y niveles de consistencia disponibles.

5.2.1 Consideraciones previas

Para realizar la evaluación de escalabilidad se deben tener en cuenta las siguientes consideraciones.

- En cada una de las pruebas se añadirá un nuevo nodo de trabajo al despliegue y se volverán a medir tanto las operaciones de lectura como las operaciones de escritura dando unos resultados comparables con la iteración anterior.
- Para la operaciones de escritura y lectura se ha utilizado el mismo trabajo de Spark en todas las arquitecturas sin ningún tipo de modificación y sin aumentar el paralelismo de las tareas.
- Los resultados de velocidad de transferencia están normalizados al numero de nodos.
- En todos los casos se utiliza un fichero de 1GB, aunque se realizaron pruebas con ficheros más grandes (con un tamaño de hasta 32GB),

se consideró que el tamaño de los ficheros era irrelevante para los resultados de la prueba.

- El sistema está configurado para un tamaño de bloque de 64MB.
- En el caso de la escritura se ha eliminado el tiempo consumido por la carga de la información a los nodos de trabajo, ya que no es un tiempo que este influido directamente por el sistema RogerFS.
- En el caso de la lectura la operación que se ha realizado, ha sido un la llamada a un método que cuenta todas las líneas que hay en el fichero, y en este caso sólo se ha tenido en cuenta el tiempo tardado a la hora de leer los datos de la base de datos.
- Los factores de replicación utilizados son uno y tres.
- En las pruebas de con factor de replicación tres, no se han incluido las pruebas con uno y con dos nodos debido a que no se pueden considera pruebas en donde el sistema se encuentre en un estado estable.
- En las pruebas con factor de replicación tres, se han tenido en cuenta dos niveles de consistencia ONE, nivel de consistencia por defecto en Cassandra, y QUORUM, nivel de consistencia recomendado para sistemas en producción, se han descartado utilizar otros niveles de consistencia debido a que estos son los más utilizados.

Todas esta consideraciones se deben tener en cuenta a la hora de analizar los resultados.

5.2.1 Conclusiones de la prueba

Una vez analizados los resultados de la prueba (veasé en 8. Resultados de la prueba de escalabilidad) que se han realizado, se han extraído las siguientes conclusiones.

- La escalabilidad lineal esta garantizada, en las pruebas realizadas apenas se notan perdidas en la velocidad de proceso cuando se añaden nodos.

- El factor de replicación y la el nivel de consistencia afectan singularmente al rendimiento del sistema.
- La velocidad de lectura se ve beneficiada con el factor de replicación como se puede observar en la siguiente imagen (Figure 18).

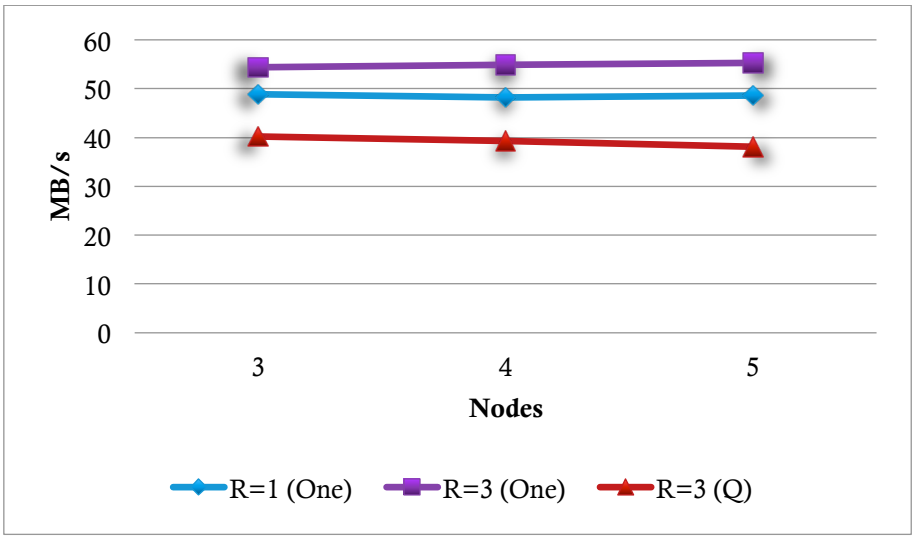

Figure 18. Comparativa de velocidad de lectura en los diferentes escenarios.

- La velocidad de escritura no depende del factor tanto del factor de replicación como de nivel de consistencia como se puede observar en la siguiente imagen (Figure 19).

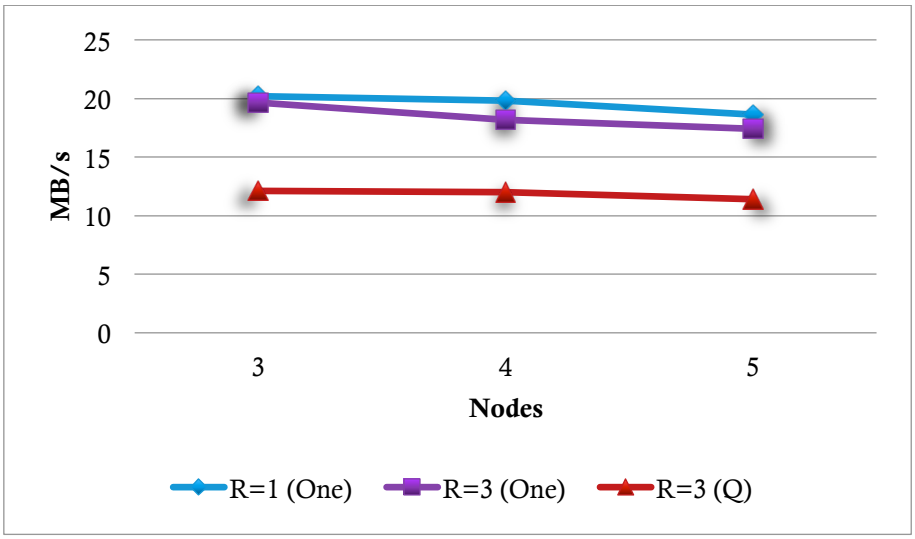

Figure 19. Comparativa de velocidad de escritura en los diferentes escenarios.

5.3 Comparativa con HDFS

Actualmente existen otros sistemas de ficheros distribuidos en el mercado, es por ello que es necesario para mostrar la viabilidad del proyecto compararse con ellos, en este caso se ha elegido HDFS, ya que actualmente es el sistema de ficheros más utilizado dentro del ecosistema Big Data.

El objetivo de esta evaluación es comparar el sistema RogerFS con otros sistemas de fichero, para comprobar el *overhead* que introduce un sistema de base de datos distribuida al ser utilizado como sistema de ficheros. A pesar de conocer de antemano que el rendimiento del sistema sea menor, las pruebas esperan que ese porcentaje sea lo suficiente escaso que pueda ser asumible para su uso de producción.

Dependiendo de el porcentaje de *overhead* introducido por el sistema, este podría ser mitigado con posible optimizaciones de código que quedan fuera del alcance de este proyecto.

- **Mayor del 40%.** El sistema no es usable y no es asumible su puesta en sistema de producción debido a que la diferencia con sistema específicos para este problema es tan grande que no compensa el uso de esta tecnología.
- **Mayor del 20%.** En este caso el sistema podría considerar listo para producción después de haber hecho tareas de optimización que nos permitieran mejorar el sistema.
- **Menor del 20%.** En este caso el sistema estaría listo para ser puesto en producción debido a que los márgenes son razonables y a que después de diferentes tarea de optimización, el sistema podría ser comparable a sistemas de uso específico.

5.3.1 Consideraciones previas

Para hacer la comparativa entre los dos sistemas se deben tener en cuenta las siguientes consideraciones:

- El despliegue estará dimensionado para cinco nodos, estos nodos no serán modificado después de cada una de las pruebas únicamente se vaciará la información almacenada en los sistemas para que esta no pueda afectar a iteraciones posteriores.
- El fichero utilizado para la prueba es un fichero de 1GB se hicieron pruebas con ficheros de tamaño mayor (hasta 32GB) y se comprobó que el tamaño de los ficheros no influye en la velocidad de escritura y lectura del sistema.
- En el sistema de RogerFS se opto por utilizar un nivel de consistencia ONE, para igualarlo con el sistema HDFS.
- Se han utilizado dos factores de replicación de uno y de tres para comprobar la diferencia del sistema en estas situaciones.
- En ambos caso se ha utilizado un tamaño de bloque de 64MB.
- La aplicaciones Spark utilizadas en ambos despliegues son equivalentes.
- En ambos casos se ha eliminado el tiempo de lectura de los ficheros en las operaciones y el tiempo de las operaciones posteriores en el caso de las lecturas.

Todas esta consideraciones se deben tener en cuenta antes de analizar los resultados.

5.3.2 Conclusiones de la prueba

Una vez analizados los resultados obtenidos de la prueba con los diferentes sistemas (veasé en 9. Resultados comparativa con HDFS) se han obtenido las siguientes conclusiones.

- Los resultados en todos los ámbitos de la prueba según los parámetros objetivo son aceptables. Ya que la perdida de rendimiento nunca ha sido superior al 30%, tal y como se puede ver en la siguiente tabla (Table 12).

Table 12. Comparativa de rendimiento frente a HDFS.

Factor de Replicación	Escrituras	Lecturas
R=1	+29,7%	+14,9%
R=3	-6,1%	+16,5%

- Los resultados son mucho mejores en el proceso de escritura que en el lectura.
- Durante las prueba se han encontrado diferente puntos de mejora que permiten mejorar el rendimiento general del sistema y mejorar esta comparativa, aunque la ejecución de los mismos quedan fuera del alcance de este proyecto.

6 Conclusiones

En este apartado se hará un repaso de las conclusiones que se han extraído durante el desarrollo del proyecto, se listarán los objetivos que se han cumplido, los problemas detectados y finalmente se expondrán las posibles mejoras con las que se puede continuar el trabajo que se ha realizado.

6.1 Objetivos

El principal objetivo del proyecto era crear un sistema de ficheros distribuido utilizando un motor de una base de datos NoSQL eficiente y capaz de ser una alternativa a otros sistemas de ficheros existentes en el mercado.

Según los datos obtenidos durante las pruebas realizadas, este objetivo ha sido cumplido, ya que el sistema permite trabajar con ficheros con un rendimiento similar a uno de los sistema de ficheros distribuido más utilizado en la actualidad. Si a esto se une la reducción significativa del stack tecnológico necesario para el despliegue y la facilidad de uso. Estaríamos frente a una solución claramente competitiva frente a las tecnologías existentes.

Uno de los objetivos de este proyecto era que la introducción del sistema fuera lo menos invasiva posible, que requiriera la mínima configuración y preferiblemente que no obligará a instalar nuevos componentes en el sistema, esto ha sido conseguido sustancialmente debido a que para usar el sistema de ficheros no es necesario instalar nada en el servidor, sólo hay que instanciar la librería en los clientes por lo que hace mucho más sencillo el trabajo.

El objetivo principal este proyecto buscaba el desarrollo de un sistema de fichero sobre una base de datos NoSQL, y aunque en este proyecto la base de datos seleccionada ha sido Cassandra, el sistema esta preparado para ser extendida con cualquier base de datos disponible, para ellos sólo se

debe extender la clase IStore que provee los método que el sistema necesita para poder ejecutar todos los comandos y dar soporte a lectura y escritura en el sistema.

Como también se ha podido comprobar en la fase de pruebas el sistema es altamente escalable y permite su funcionamiento en diferentes modos de consistencia, lo cual permite que el usuario sea estricto con la consistencia del sistema o en los casos que la información no sea tan critica puede ajustar este nivel de consistencia primando el rendimiento del sistema.

El sistema RogerFS es un sistema altamente escalable, extensible, fácil de desplegar que puede competir con otros sistemas de ficheros distribuidos que se encuentran en el mercado.

6.2 Problemas encontrados

Durante el desarrollo del proyecto han ido apareciendo diferentes problemas causados por la elección de diferentes tecnologías para la implementación del proyecto, parte de las futuras mejoras serán sustituir algunas tecnologías seleccionadas y en otro caso contribuir en los proyectos para mejorar el soporte de ciertas funcionalidades.

6.2.1 SBT

El gestor de dependencias seleccionado para el proyecto fue SBT, esta elección se produjo esencialmente por dos motivos, el primero que es el gestor de dependencias recomendado para proyecto Scala, y segundo que es el único que soporta de forma nativa proyectos mixto Java/Scala. Sin embargo, durante el desarrollo del proyecto este gestor de dependencias se ha mostrado en muchos casos en un problema, debido a que el soporte de proyectos multi-módulo es anti-intuitivo, el crear diferentes niveles de dependencias ha sido un proceso laborioso que ha pasado por diferentes niveles de documentación, foros y revisión de proyectos que utilizan este gestor de archivos. El rendimiento del sistema a la hora de hacer tareas rutinarias como la de compilación y la del lanzamiento de pruebas, no era

el esperado. Con el conocimiento actual, probablemente el proyecto sería desarrollado utilizando otro gestor de dependencias, como por ejemplo, Apache Maven.

6.2.2 Spark Cassandra Connector

Otra librería que ha generado problemas, ha sido el Driver de Spark para Cassandra implementado por la empresa DataStax, en este caso los problemas principalmente han sido provocados por la falta de soporte de ciertas características de la base de datos. Como por ejemplo las columnas estáticas o el uso de índices secundarios. En este caso, la gente encargada del proyecto era consciente de este problema y tiene prevista la inclusión de estas características en siguientes versiones.

6.2.3 Despliegue de pruebas

Por último, no se crearon scripts de despliegue para lanzar la diferentes pruebas realizadas sobre el sistema. Esto hizo que las pruebas duraran más tiempo de lo esperado, este problema de planificación ha afectado sobre todo al tiempo que se han mantenido las máquinas reservadas.

6.3 Mejoras y trabajos futuros

En esta sección se hará un repaso de todos las mejoras propuestas al sistema y los trabajos que se pueden realizar en siguiente versiones de proyecto.

6.3.1 Ampliación del soporte a base de datos NoSQL

Aunque el sistema actual sólo soporta Cassandra, debido su capacidades de extensión sería muy sencillo ampliar el soporte a otras bases de datos del sector, como por ejemplo, MongoDB, CouchDB o Aerospike.

Por otro lado, también podrías se muy interesante incluir soporte a HDFS desde la propia librería de esta forma sería sencillo poder extender las capacidades del sistema de ficheros con las características que se vayan

añadiendo al sistema y permitiría comparar estas funcionalidades con otros motores.

6.3.2 Crear una abstracción de RDD para IStore

Una gran mejora de diseño que se podría añadir al sistema sería incluir una abstracción de RDD o de Dataframe a la IStore, para ello se necesitaría añadir los métodos necesarios para que las consultas puedan ser independientes del sistema que este utilizando.

Esta forma de trabajo daría muchas más potencia al motor y permitiría seguir evolucionando la funcionalidades ofrecidas al usuario sin necesidad de re-implementar los driver para cada una de las plataformas.

6.3.3 Optimizar el sistema para ficheros de líneas

La mayoría de los ficheros que son almacenados en esto sistemas son ficheros divididos por líneas con CSV, TSV o JSON divididos por líneas, en este caso creemos que el motor puede hacer ciertas optimizaciones en el sistema cuando ocurran estos casos:

- **No dividir el fichero a mitad de una línea.** Cuando el fichero ocupa más de un bloque, actualmente, la división es arbitraría lo cual provoca en que para poder recuperar ficheros por líneas sea necesario realizar un procesado que reconstruya las líneas almacenadas, sin embargo, si se realizan esta optimizaciones se puede garantizar que ninguna línea quede divida en dos bloques por lo que la reconstrucción en este caso no será necesaria.
- **Optimización para datos semi-estructurados.** En este caso cuando se declare un esquema concreto en el sistema se podría implementar ese esquema directamente con la base de datos lo que nos permitiría realizar consulta más potentes y utilizar ciertas librerías de Spark para las que actualmente no hay soporte, como SparkSQL o GraphX.

6.3.4 Soporte para SparkSQL

Dar soporte para el uso de Dataframes usando la abstracción de IStore y la posibilidad de soportar un esquema asociado a los ficheros. Esto nos permitiría soportar consultas SQL sobre nuestro almacén de datos, permitiendo a los usuarios mayor flexibilidad para poder trabajar con los datos del sistema.

6.3.5 Soporte a Apache Flink

Flink es un sistema de procesamiento distribuido más moderno que Apache Spark y que poco a poco esta ganando fuerza dentro del ecosistema Big Data. Para poder soportar diferentes motores de procesamiento sería necesario crear una pequeña abstracción para poder separar la operaciones necesarias en cada una de las partes. Sin embargo, daría al sistema la capacidad de adaptarse con mayor facilidad a diferentes escenarios y a futuras tecnologías.

7 Planificación y presupuesto

En este apartado se darán detalles sobre la planificación y el presupuesto requerido por el proyecto.

7.1 Planificación

En la siguiente figura (Figure 20) se puede ver un diagrama de Gantt con todas las tareas que se han llevado durante el proyecto y las horas dedicadas a cada una de ellas a continuación se van a desglosar dichas tareas y se va especificar el alcance de cada una de ellas.

Table 13. Planificación del proyecto.

ID	Tarea	Duración	Descripción
1	**Inicio del proyecto**		
2	**Diseño**		
3	Recogida de requisitos	1 d	Captación de los requisitos con la función que debe cubrir el proyecto.
4	Diseño de objetivos	2 d	Definición de objetivos de calidad y de rendimiento del sistema.
5	Diseño de arquitectura	2 d	Definición de la arquitectura del sistema y de sus componentes.

6	Diseño de pruebas	1 d	Diseño de las pruebas necesarias que garantizan el cumplimiento de los objetivos.
7	**Desarrollo**		
8	Core	3 d	Implementación del Core de la arquitectura.
9	Api Java	3 d	Creación del Api de lectura y escritura de Streams.
10	Driver de Cassandra	2 d	Desarrollo del driver para la base de datos Cassandra.
11	Integración Spark	4 d	Ampliación de la funcionalidad del Core para añadir el modulo de procesamiento Spark.
12	Aplicación Shell	2 d	Creación de la aplicaciones por línea de comandos.
13	Test de integración	3 d	
14	**Pruebas**		
15	Despliegue infrastructura	2 d	Despliegue de la arquitectura necesaria para el proyecto.
16	Ejecución de pruebas	3 d	Ejecución de la pruebas del sistema.
17	Análisis de resultados	1 d	Recogida y análisis previó de los resultados obtenidos.

18	**Documentación**		
19	Estado del arte	4 d	Documentación del estado de la cuestión.
20	Arquitectura y diseño	10 d	Documentación del diseño y la arquitectura del sistema.
21	Pruebas	5 d	Documentación del desarrollo y las conclusiones extraídas de las pruebas efectuadas.

7.2 Presupuesto

Las dos grandes partidas presupuestarias de este proyecto son el personal y el abastecimiento de maquinas por parte de nuestro proveedor Cloud, en este caso Amazon AWS. Para los recursos humanos se ha tenido en cuenta como base salarial la de un desarrollador software senior con un coste por hora de 65€. Para los recursos Cloud, se ha utilizados el desglose proporcionado por la propia compañía. En la siguiente tabla se hace un desglose del presupuesto.

Table 14. Presupuesto del proyecto.

Partida	Unidades	Coste Unitario	Total
Diseño	1,2 semanas	65€/dia	**3.120,00€**
Desarrollo	3,4 semanas	65€/dia	**8.840,00€**
Pruebas	1,2 semanas	65€/dia	**3.120,00€**
Documentación	3,8 semanas	65€/dia	**9.980,00€**

Cloud	415horas	0,266€/hora	**110,39€**
Total			**25.070,39€**

#	Traits	Title	Given Work	Given Earliest Start	Resources	Predecessors
0		**RogerFs**		**1 Jun 2015**	**DES**	
1		Inicio del projecto		1 Jun 2015		
2		**Diseño**			**DES**	**1**
3		Recogida de requisitos	1 day		DES	
4		Diseño de objetivos	2 days		DES	3
5		Diseño de arquitectura	2 days		DES	4
6		Diseño de pruebas	1 day		DES	5
7		**Desarrollo**			**DES**	**2**
8		Core	3 days		DES	
9		Api Java	3 days		DES	
10		Driver de Cassandra	2 days		DES	8; 9
11		Integración Spark	4 days		DES	8; 9
12		Aplicación Shell	2 days		DES	11
13		Test de integración	3 days		DES	10; 11; 12
14		**Pruebas**			**DES**	**7**
15		Despliegue infrastructura	2 days		DES	
16		Ejecución de pruebas	3 days		DES	15
17		Análisis de resultados	1 day		DES	16
18		**Documentación**			**DES**	
19		Estado del arte	4 days		DES	2
20		Arquitectura y diseño	10 days		DES	7
21		Pruebas	5 days		DES	14

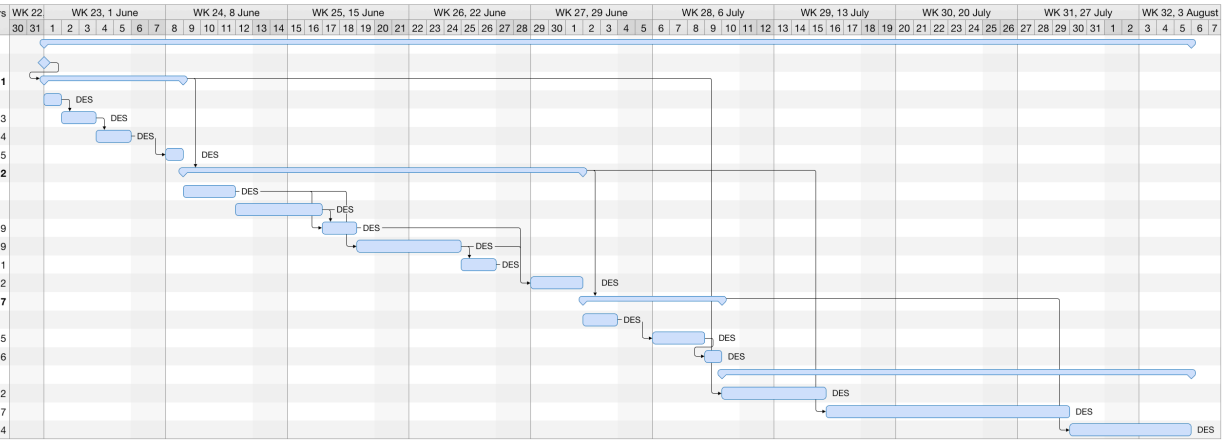

Figure 20. Planificación del proyecto.

8 Resultados de la prueba de escalabilidad

En esta sección se van a presentar todos los resultados de la prueba, en los diferentes escenarios planteados.

8.1 Replicación 1 y consistencia ONE

Table 15. Evaluación de escalabilidad: escrituras con replicación 1 y consistencia ONE.

Nodes	T1	T2	T3	T4	T5	Avg (s)	Desv.	Avg (MB/s)
1	41,32	42,05	41,44	41,65	41,70	41,63	0,28	24,60
2	24,16	23,73	23,36	23,87	23,56	23,74	0,30	21,57
3	16,93	16,65	17,28	16,63	17,05	16,91	0,28	20,19
4	13,34	13,22	12,74	12,82	12,41	12,91	0,38	19,84
5	10,87	10,96	11,23	11,13	10,78	10,99	0,18	18,63

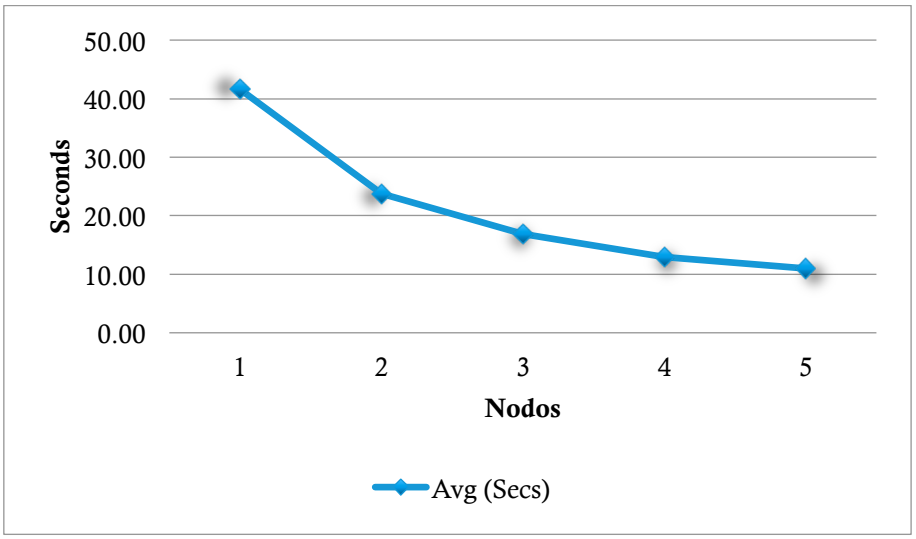

Figure 21. Tiempo medio de escritura con replicación 1 y consistencia ONE.

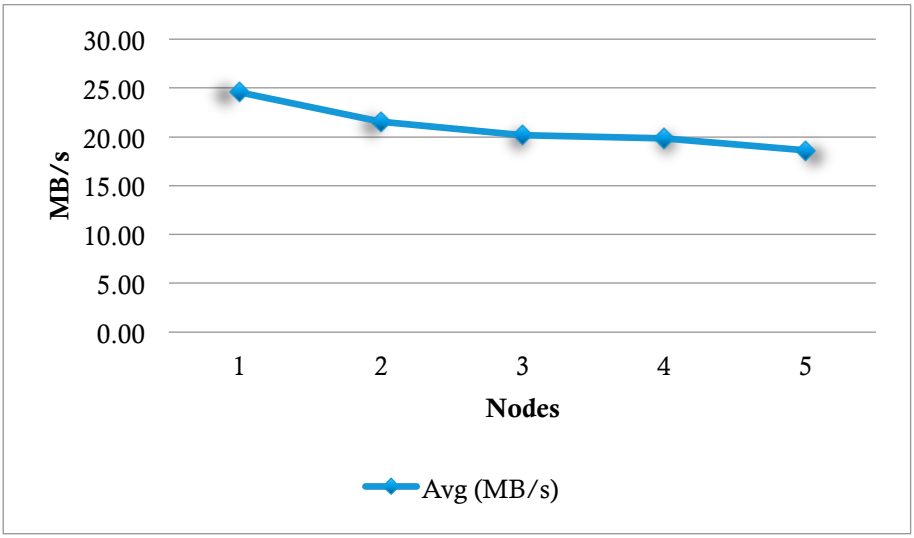

Figure 22. Velocidad de escritura con replicación 1 y consistencia ONE.

Table 16. Evaluación de escalabilidad: lecturas con replicación 1 y consistencia ONE.

Nodes	T1	T2	T3	T4	T5	Avg (s)	Desv.	Avg (MB/s)
1	17,13	17,23	16,98	17,02	17,27	17,13	0,13	59,79
2	9,14	9,32	9,22	9,67	9,46	9,36	0,21	54,69
3	7,19	6,95	6,88	7,03	6,89	6,99	0,13	48,85
4	5,25	5,29	5,82	5,08	5,13	5,31	0,30	48,17
5	4,23	4,36	4,14	4,12	4,21	4,21	0,09	48,62

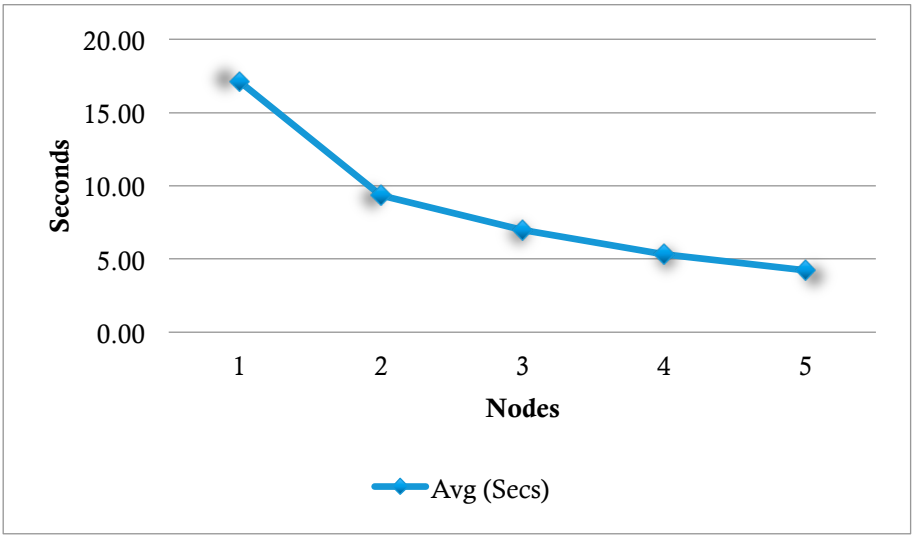

Figure 23. Tiempo medio de escritura con replicación 1 y consistencia ONE.

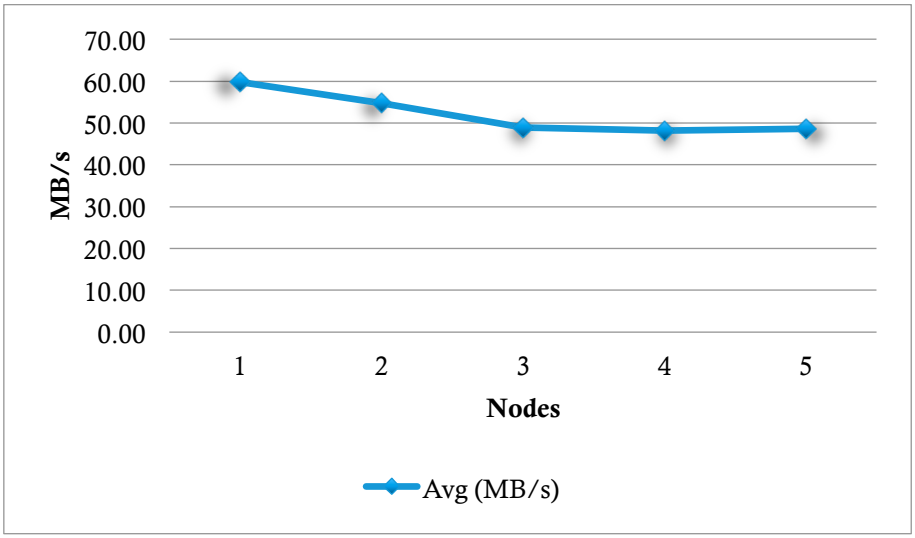

Figure 24. Velocidad de escritura con replicación 1 y consistencia ONE.

8.2 Replicación 3 y consistencia ONE

Table 17. Evaluación de escalabilidad: escrituras con replicación 3 y consistencia ONE.

Nodes	T1	T2	T3	T4	T5	Avg (s)	Desv.	Avg (MB/s)
1	--	--	--	--	--	--	--	--
2	--	--	--	--	--	--	--	--
3	17,23	17,15	17,56	17,65	17,23	17,36	0,22	19,66
4	14,22	14,13	13,98	13,87	14,12	14,06	0,14	18,20
5	11,65	11,87	11,88	11,59	11,83	11,76	0,13	17,41

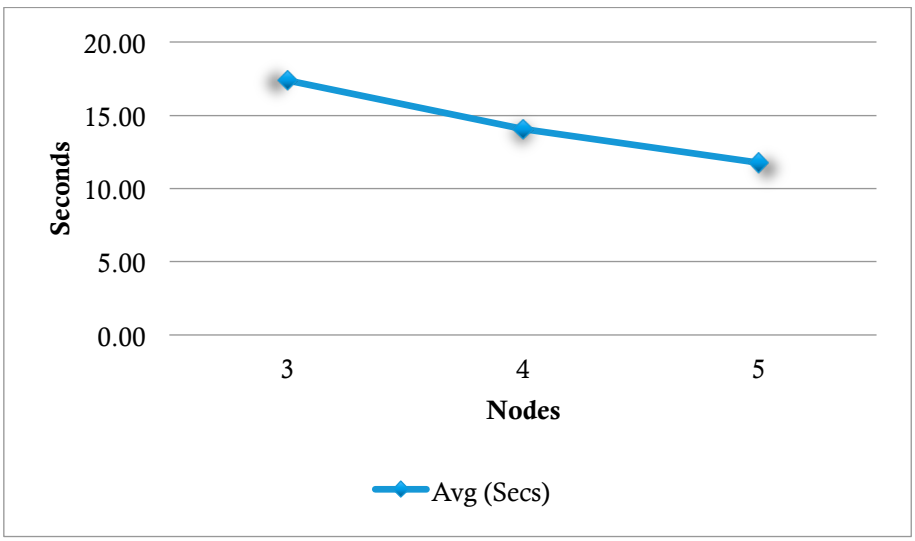

Figure 25. Tiempo medio de escritura con replicación 3 y consistencia ONE.

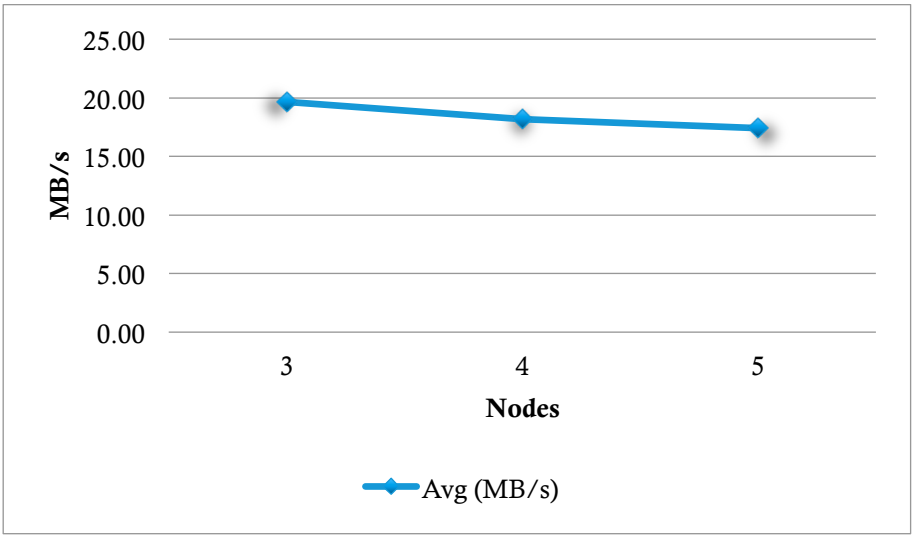

Figure 26. Velocidad de escritura con replicación 3 y consistencia ONE.

Table 18. Evaluación de escalabilidad: lecturas con replicación 3 y consistencia ONE.

Nodes	T1	T2	T3	T4	T5	Avg (s)	Desv.	Avg (MB/s)
1	--	--	--	--	--	--	--	--
2	--	--	--	--	--	--	--	--
3	6,05	6,42	6,33	6,21	6,36	6,27	0,15	54,40
4	4,99	4,68	4,79	4,52	4,33	4,66	0,25	54,91
5	3,74	3,89	3,57	3,77	3,56	3,71	0,14	55,26

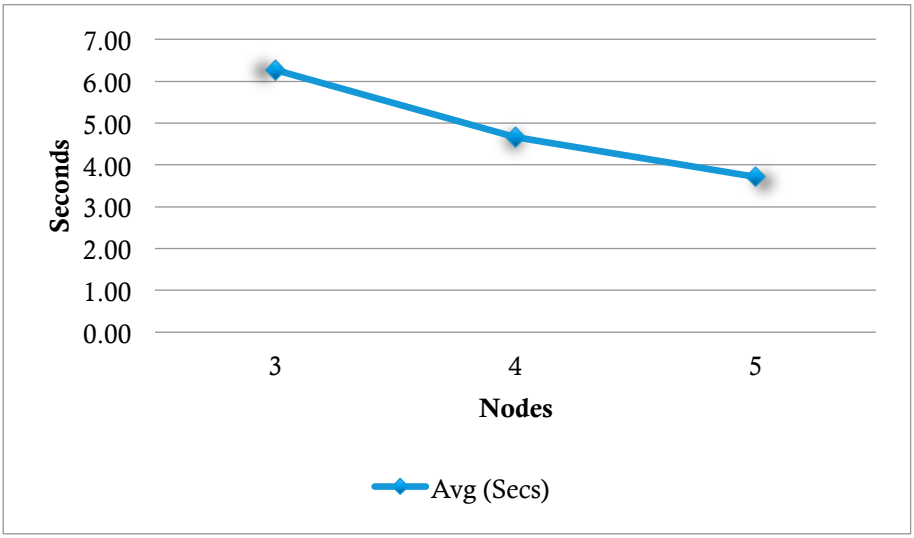

Figure 27. Tiempo medio de lectura con replicación 3 y consistencia ONE.

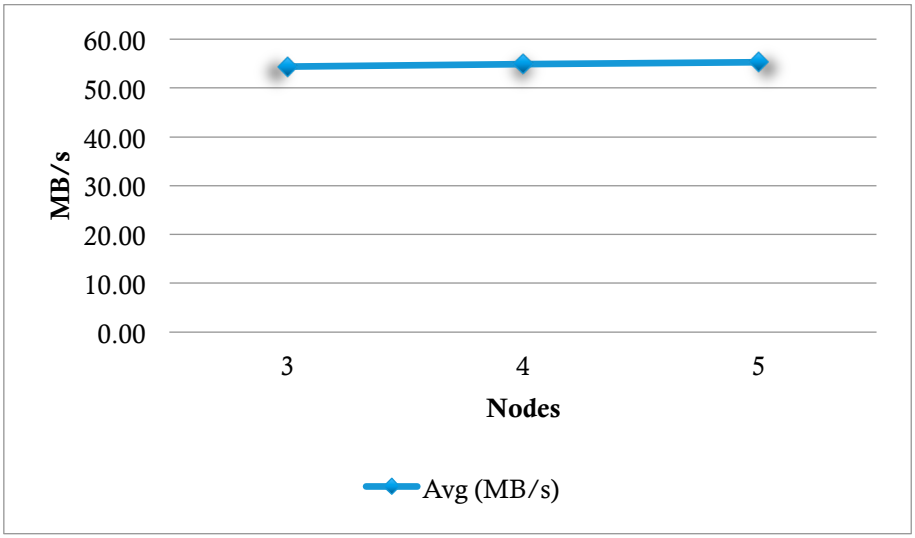

Figure 28. Velocidad de lectura con replicación 3 y consistencia ONE.

8.3 Replicación 3 y consistencia QUORUM

Table 19. Evaluación de escalabilidad: escrituras con replicación 3 y consistencia QUORUM.

Nodes	T1	T2	T3	T4	T5	Avg (s)	Desv.	Avg (MB/s)
1	--	--	--	--	--	--	--	--
2	--	--	--	--	--	--	--	--
3	28,34	28,09	27,76	27,96	28,43	28,12	0,27	12,14
4	21,09	21,45	21,34	21,63	21,22	21,35	0,21	11,99
5	17,81	18,12	18,22	18,05	17,65	17,97	0,23	11,40

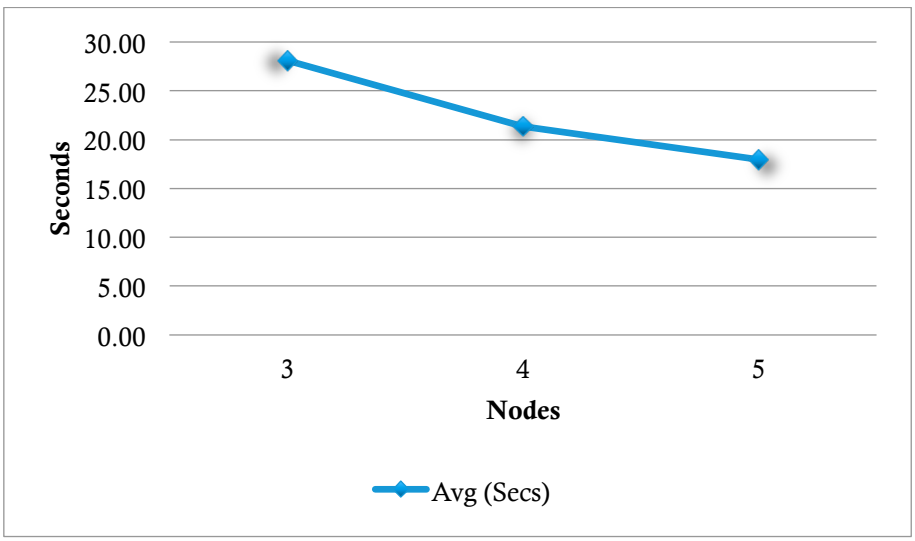

Figure 29. Tiempo medio de escritura con replicación 3 y nivel de consistencia QUORUM.

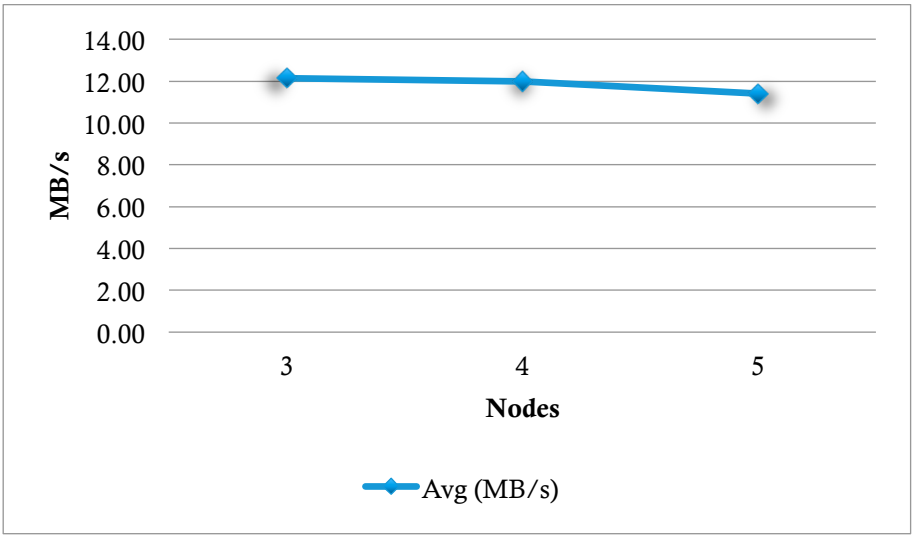

Figure 30. Velocidad de escritura con replicación 3 y nivel de consistencia QUORUM.

Table 20. Evaluación de escalabilidad: lecturas con replicación 3 y consistencia QUORUM.

Nodes	T1	T2	T3	T4	T5	Avg (s)	Desv.	Avg (MB/s)
1	--	--	--	--	--	--	--	--
2	--	--	--	--	--	--	--	--
3	8,97	8,11	8,61	8,10	8,66	8,49	0,38	40,20
4	6,14	6,76	6,50	6,43	6,77	6,52	0,26	39,26
5	5,45	5,13	5,07	5,76	5,49	5,38	0,28	38,07

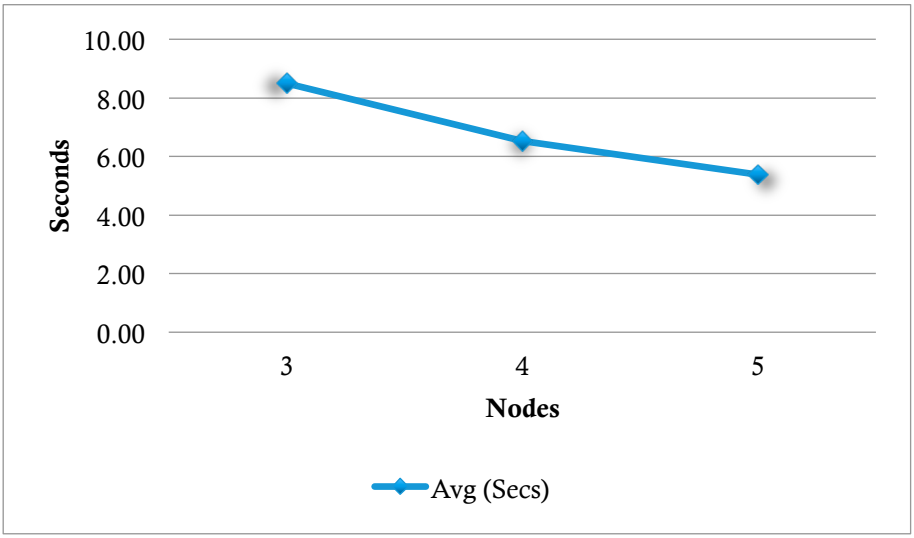

Figure 31. Tiempo medio de lectura con replicación 3 y consistencia QUORUM.

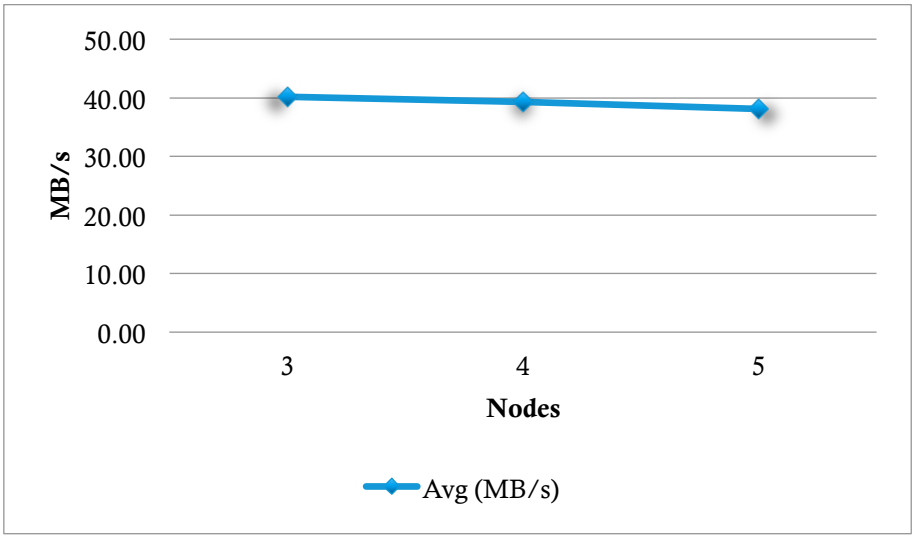

Figure 32. Velocidad de lectura con replicación 3 y consistencia QUORUM.

9 Resultados comparativa con HDFS

En esta sección se van a presentar todos los resultados de la prueba, en los diferentes escenarios planteados.

Table 21. Comparación del sistema: lecturas.

Test	T1	T2	T3	T4	T5	Avg (s)	Desv.	Avg (MB/s)
RFS[30] (R=1)	4,23	4,36	4,14	4,12	4,21	4,21	0,09	48,62
HDFS (R=1)	3,25	3,56	3,88	3,67	3,77	3,63	0,24	56,48
RFS (R=3)	3,74	3,89	3,57	3,77	3,56	3,71	0,14	55,26
HDFS (R=3)	3,17	3,02	3,12	3,10	3,06	3,09	0,06	66,19

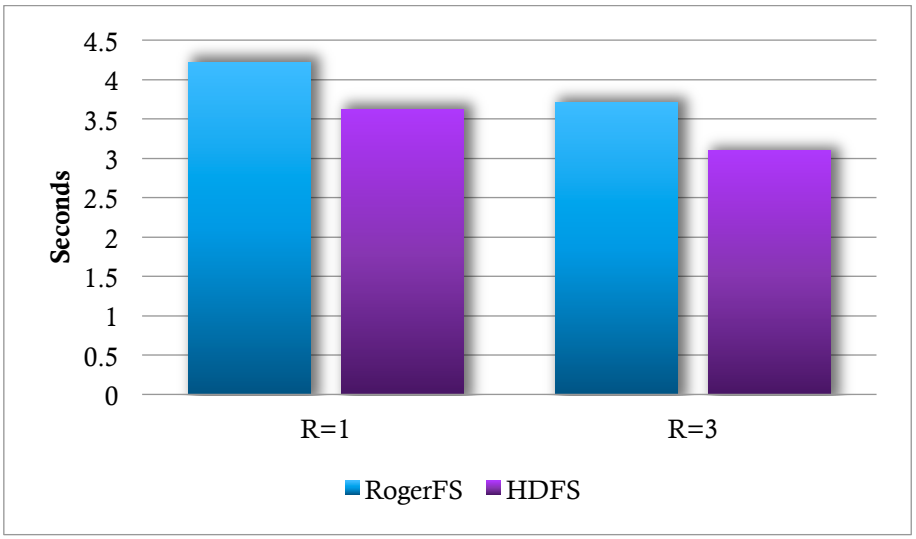

Figure 33. Tiempo medio de lectura.

[30] RogerFS con replicación uno.

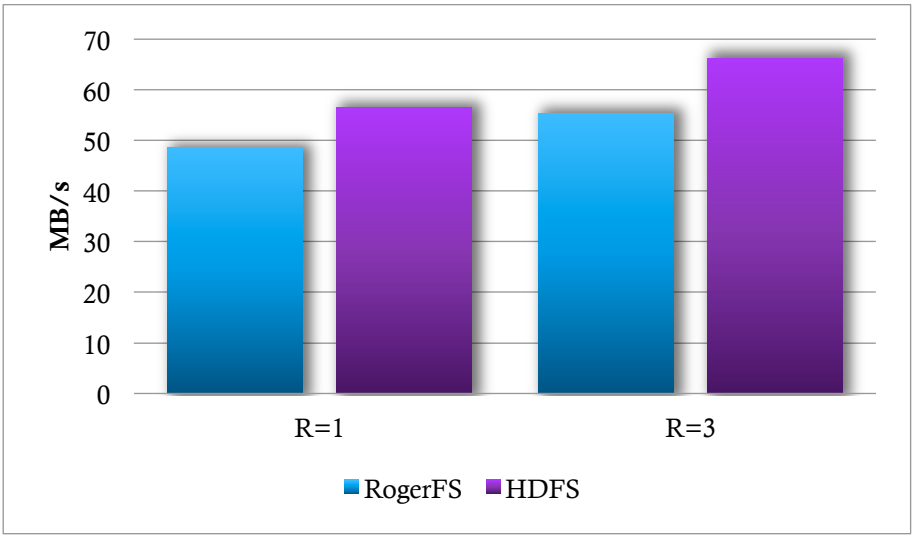

Figure 34. Velocidad de lectura.

Table 22. Comparación del sistema: escrituras.

Test	T1	T2	T3	T4	T5	Avg (s)	Desv.	Avg (MB/s)
RFS[31] (R=1)	10,87	10,96	11,23	11,13	10,78	10,99	0,18	18,63
HDFS (R=1)	7,80	7,36	7,65	7,79	7,88	7,70	0,21	26,61
RFS (R=3)	11,65	11,87	11,88	11,59	11,83	11,76	0,13	17,41
HDFS (R=3)	12,45	12,31	12,76	12,43	12,51	12,49	0,17	16,39

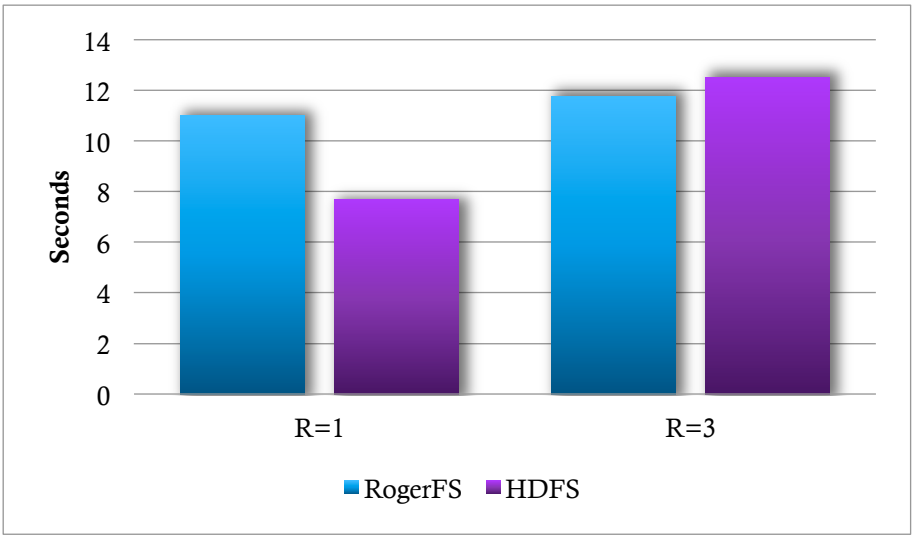

Figure 35. Tiempo medio de escritura.

[31] RogerFS con replicación uno.

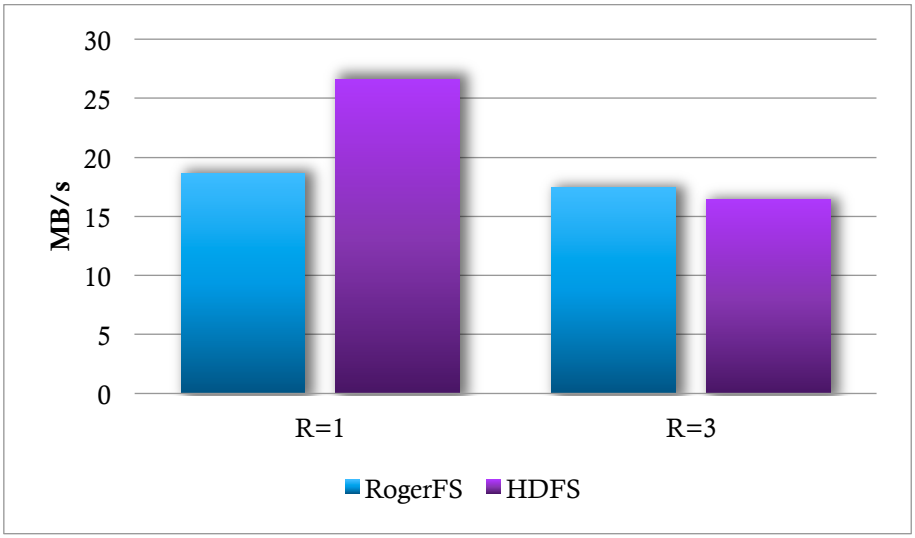

Figure 36. Velocidad de escritura.

10 Bibliografía

Cassandra. (2015). *Apache Cassandra*. Retrieved 24 de Abril de 2015 from Apache Cassandra: http://cassandra.apache.org/

Cassandra CQL. (Mayo de 2015). *Cassandra Query Language (CQL) v3.2.0*. Retrieved 10 de Mayo de 2015 from Cassandra: https://cassandra.apache.org/doc/cql3/CQL.html

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M. , et al. (2006). Bigtable: A Distributed Storage System for Structured Data. Seattle: OSDI.

Cloudera Impala. (Mayo de 2015). *Impala*. Retrieved 10 de Mayo de 2015 from Impala: http://impala.io/

Dean, J., & Ghemawat, S. (2004). MapReduce: simplified data processing on large clusters. *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco: USENIX.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., et al. (2007). Dynamo: Amazon's Highly Available Key-value Store . Stevenson: SOSP.

Deutsch, P. (1995). *The Eight Fallacies of Distributed Computing*. Retrieved 27 de August de 2015 from Oracle Blog: https://blogs.oracle.com/jag/resource/Fallacies.html

Drill. (Mayo de 2015). *Apache Drill*. Retrieved 10 de Mayo de 2015 from Apache Drill: http://drill.apache.org/

Flink. (Mayo de 2015). *Apache Flink*. Retrieved 10 de Mayo de 2015 from Apache Flink: https://flink.apache.org/

Giraph. (Mayo de 2015). *Apache Giraph*. Retrieved 2 de Mayo de 2015 from Apache Giraph: http://giraph.apache.org/

Google. (Marzo de 2015). *Google Java Style*. Retrieved 17 de Mayo de 2015 from Google Java Style: https://google-styleguide.googlecode.com/svn/trunk/javaguide.html

Hadoop. (23 de Abril de 2015). *Apache Hadoop*. Retrieved 24 de Abril de 2015 from https://hadoop.apache.org/

HBASE. (Abril de 2015). *HBase*. Retrieved 24 de Abril de 2015 from HBase: http://hbase.apache.org/

Hewitt, E. (2010). *Cassandra: The Definitive Guide*. (M. Loukides, Ed.) O'Reilly Media, Inc.

Hive. (Mayo de 2015). *Apache Hive*. Retrieved 2 de Mayo de 2015 from Apache Hive: https://hive.apache.org/

Isard, M., Budiu, M., Yu, Y., Birrell, A., & Fetterly, D. (2007). Dryad: distributed data-parallel programs from sequential building blocks. *Dryad: distributed data-parallel programs from sequential building blocks*. Lisboa: EuroSys.

Mahout. (Mayo de 2015). *Apache Mahout*. Retrieved 2 de Mayo de 2015 from Apache Mahout: http://mahout.apache.org/

Sanjay , G., Howard , G., & Shun-Tak , L. (2003). The Google File System . *19th ACM Symposium on Operating Systems Principles*. Lake George, NY: ACM.

SBT. (Mayo de 2015). *SBT*. Retrieved 17 de Mayo de 2015 from SBT: http://www.scala-sbt.org/

Scala-Lang. (Mayo de 2015). *Scala Style Guide*. Retrieved 17 de Mayo de 2015 from Scala Documentation: http://docs.scala-lang.org/style/

Spark. (Abril de 2015). *Apache Spark*. Retrieved 24 de Abril de 2015 from Apache Spark: http://spark.apache.org/

Stoica, I., Morris, R., Karger, D., Kaashoek, M., & Balakrishnan, H. (2001). Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In ACM (Ed.), *SIGCOMM*. San Diego: ACM.

Tez. (Mayo de 2015). *Apache Tez*. Retrieved 10 de Mayo de 2015 from Apache Tez: http://tez.apache.org/

Thrift. (Mayo de 2015). *Apache Thrift*. Retrieved 10 de Mayo de 2015 from Apache Thrift: https://thrift.apache.org/

Typesafe. (Mayo de 2015). *Scala*. Retrieved 10 de Mayo de 2015 from Scala: http://scala-lang.org/

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., et al. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *NSDI*. San Jose: USENIX.

Zaharia, M., Das, T., Li, H., Shenker, S., & Stoica, I. (2012). Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters. *HotCloud'12*. Boston: USENIX Association.

Zookeeper. (Mayo de 2015). *Apache Zookeeper*. Retrieved 2 de Mayo de 2015 from Apache Zookeeper: https://zookeeper.apache.org/